caCORE EVS API 4.0 Technical Guide

Center for Bioinformatics

January 28, 2008

Credits and Resources

caCORE EVS Development and Management Teams		
Development	***Technical Guide***	***Program Management***
Johnita Beasley[1]	Johnita Beasley[1]	Bill Britton[7]
Steve Hunter[4]	Wendy Erickson-Hirons[3]	Peter Covitz[2]
Norval Johnson[7]	Frank Hartel[2]	Frank Hartel[2]
Sriram Kalyanasundaram[7]	Shaziya Muhsin[1]	Charles Griffin[4]
Doug Kanoza[7]	Kim Ong[5]	Jason Lucas[5]
Alan Klink[7]	Konrad Rokicki[1]	Krishnakant Shanbhag[2]
Shaziya Muhsin[1]	Tracy Safran[1]	Denise Warzel[2]
Kim Ong[5]		
John Park[6]		
Konrad Rokicki[1]		
Tracy Safran[1]		
Ye Wu[1]		
Robert Wynne[6]		
Robert Wysong[7]		
David Yee[5]		

[1] Science Application International Corporation (SAIC)	[3] ScenPro, Inc.
[2] National Cancer Institute Center for Bioinformatics (NCICB)	[4]Ekagra
[5]Northrup-Grumman	[6]Lockeed Martin
[7]Terpsys	

Contacts and Support	
NCICB Application Support	http://ncicb.nci.nih.gov/NCICB/support Telephone: 301-451-4384 Toll free: 888-478-4423

LISTSERV Facilities Pertinent to the caCORE EVS		
LISTSERV	***URL***	***Name***
NCI EVS Listserv	https://list.nih.gov/archives/ncievs-l.html	NCI Vocabulary Services Information
caCORE_SDK_ Developers	https://list.nih.gov/archives/cacore_sdk_dev-l.html	caCORE SDK Developers Discussion Forum
caCORE_SDK_ Users	https://list.nih.gov/archives/cacore_sdk_users-l.html	caCORE SDK Users Discussion Forum

Release Schedule

This guide has been updated for the caCORE EVS 4.0 release. It may be updated between releases if errors or omissions are found. The current document refers to the 4.0 version of caCORE EVS, released in November 2007 by the NCICB.

Table of Contents

Chapter 1 Using This Guide

This chapter introduces you to the caCORE EVS Technical Guide.

Topics in this chapter include:

Purpose

The *caCORE EVS API 4.0 Technical Guide* describes the Enterprise Vocabulary Services (EVS) component of the Cancer Common Ontologic Representation Environment (caCORE). caCORE is an open-source standards-based semantics-computing environment and tool set created by the National Cancer Institute (NCI) Center for Biomedical Informatics and Information Technology (CBIIT). EVS is a set of services and resources that address the NCI's needs for a controlled vocabulary. It provides the semantic base upon which the data semantics of caCORE depend.

This guide describes:

- the purpose, architecture and components of caCORE EVS.
- the APIs for accessing the caCORE EVS system including Java, Web services, and XML-HTTP.
- the API providing direct remote access to the native LexBIG Service Layer.
- an overview of UML.

Release Schedule

This guide is updated for each caCORE EVS release. It may be updated between releases if errors or omissions are found. The current document refers to the 4.0 version of caCORE EVS, which was released in November 2007 by the NCI CBIIT (formerly the National Cancer Institute Center for Bioinformatics (NCICB).

Audience

The primary audience of this guide is the application developer who wants to learn about the architecture and use and/or access the caCORE EVS APIs. caCORE EVS is generated using the caCORE Software Development Kit (SDK). For more information, see the caCORE SDK 4.0 Developer's Guide. This guide assumes that you are familiar with the Java programming language and/or other programming languages, database concepts, and the Internet. If you intend to use caCORE EVS resources in software applications, it assumes that you have experience with building and using complex data systems. Neither caCORE EVS nor this documentation is intended for "end" users, such as individual health

professionals or members of the general public, unless they are also software developers.

Additional caCORE Documentation

- The caCORE EVS 4.0 Release Notes contain a description of the end user tool enhancements and bug fixes included in this release.
- The caCORE EVS 4.0 JavaDocs contain the current caCORE EVS API specification.
- The caCORE SDK 4.0 Developer's Guide contains detailed instruction on the use of the SDK and how it aids in creating a caCORE-like software system.

Organization of This Guide

This brief overview explains what you will find in each section of this guide.

- Chapter 1, this chapter, provides an overview of this guide.
- Chapter 2 provides an overview of the Enterprise Vocabulary Services (EVS) project.
- Chapter 3 provides an overview of the NCICB caCORE infrastructure.
- Chapter 4 describes the architecture of the caCORE EVS.
- Chapter 5 describes the caCORE EVS API, the service interface layer provided by the EVS API architecture and gives examples of how to use the EVS API. It also describes the distributed LexBIG API.
- Appendix A provides a list of references used to produce this guide or referred to within the text.
- Appendix B provides two additional code examples.

Document Text Conventions

The following table (Table 1-1) shows various typefaces to differentiate between regular text and menu commands, keyboard keys, tool bar buttons, dialog box options, and text that you type. This illustrates how text conventions are represented in this manual:

Convention	***Description***
Notes	**Notes:** Notes are enclosed for emphasis
Bold	Bold type is used for emphasis, buttons, or tabs to select on windows, and names of dialog boxes.
TEXT IN SMALL CAPS	TEXT IN SMALL CAPS is used for keyboard keys that you press (for example, ALT+F4)
Text in italics	Italics are used to reference other documents, sections, figures, and tables.
`Special typestyle`	Special typestyle is used for filenames, directory names, commands, file listings, and anything that would appear in a Java program, such as methods, variables, and classes.

Convention	***Description***
Bold italics typestyle	Bold italics is used for information the user needs to enter
{ }	Curly brackets are used for replaceable items (for example, replace {home directory} with its proper value such as C:\caadapter).

Table 1-1 Document text conventions

Chapter 2 Enterprise Vocabulary Services and LexBIG

This chapter provides an overview of the Enterprise Vocabulary Services (EVS) project.

Topics in this chapter include:

Introduction

The Enterprise Vocabulary Services (EVS) project is a collaborative effort of the NCI Center for Bioinformatics (NCICB) and the NCI Office of Communications. Controlled vocabularies are important to any application involving electronic data sharing. Two areas where the need is perhaps most apparent are clinical trials data collection and reporting and more generally, data annotation of any kind. The *NCI Thesaurus* is a biomedical thesaurus developed by EVS in response to a need for consistent shared vocabularies among the various projects and initiatives at the NCI as well as in the entire cancer research community. The EVS project also produces the *NCI Metathesaurus*, which is based on NLM's Unified Medical Language System Metathesaurus (UMLS) supplemented with additional cancer-centric vocabulary.

A critical need served by the EVS is the provision of a well-designed ontology covering cancer science. Such an ontology is required for data annotation, inferencing and other functions. The data to be annotated might be anything from genomic sequences to case report forms to cancer image data. The NCI Thesaurus covers all of these domains. A few of the specialties it includes are pertinent to disease, biomedical instrumentation, anatomical structure, and gene/protein information. The NCI Thesaurus is updated monthly to keep up with developments in cancer science.

The NCI Thesaurus is implemented as a Description Logic vocabulary and, as such, is a self-contained and logically consistent terminology. Unlike the NCI Thesaurus, the purpose of the NCI Metathesaurus is *not* to provide unequivocal or even necessarily consistent definitions. The purpose of the NCI Metathesaurus, like the UMLS Metathesaurus, is to provide mappings of terms across vocabularies. The caCORE EVS interfaces, discussed later in this guide, provide access to both the NCI Thesaurus and the NCI Metathesaurus.

In the following sections, a brief overview of the UMLS Metathesaurus is provided, upon which the NCI Metathesaurus is based. This is followed by a short discussion of description logic, its role in the area of knowledge representation, and its implementation in the NCI Thesaurus.

The UMLS Metathesaurus

The NCI Metathesaurus is based on the UMLS Metathesaurus, supplemented with additional cancer-centric vocabulary. Excellent documentation on the UMLS is available at the UMLS Knowledge Sources web site.

A brief overview of the UMLS Metathesaurus is included here, but it is strongly recommended that users who wish to gain a deeper understanding refer to the above web site. Only those features of the UMLS Metathesaurus that are relevant to accessing the NCI Metathesaurus are described here.

The UMLS Metathesaurus is a unifying database of concepts that brings together terms occurring in over 100 different controlled vocabularies used in biomedicine. When adding terms to the Metathesaurus, the UMLS philosophy has been to preserve all of the original meanings, attributes, and relationships defined for those terms in the source vocabularies, and to retain explicit source information as well. In addition, the UMLS editors add basic information about each concept and introduce new associations that help to establish synonymy and other relationships among concepts from different sources.

Given the very large number of related vocabularies incorporated in the Metathesaurus, there are instances where the same concept may be known by many different names, as well as instances where the same names are intended to convey different concepts. To avoid ambiguity, the UMLS employs an elaborate indexing system, the central kingpin of which is the *concept unique identifier* (CUI). Similarly, each unique concept name or string in the Metathesaurus has a string unique identifier (SUI).

In cases where the same string is associated with multiple concepts, a numeric tag is appended to that string to render it unique as well as to reflect its multiplicity. In addition, the UMLS Metathesaurus editors may create an alternative name for the concept that is more indicative of its intended interpretation. In these cases, all three names for the concept are preserved.

Several types of relationships are defined in the UMLS Metathesaurus, and four of these are captured by the NCI Metaphrase interface:

- Broader (RB) - The related concept has a more general meaning.
- Narrower (RN) - The related concept has a more specific meaning.
- Synonym (SY) - The two concepts are synonymous.
- Other related (RO) - The relationship is not specified but is something other than synonymous, narrower or broader.

The UMLS *Semantic Network* is an independent construct whose purpose is to provide consistent categorization for all concepts contained in the UMLS Metathesaurus, and to define a useful set of relationships among these concepts. As of the 2005AC release, the Semantic Network defined a set of 135 basic semantic types or categories that could be assigned to these concepts and 54 relationships that could hold among these types.

The major groupings of semantic types include organisms, anatomical structures, biologic function, chemicals, events, physical objects, and concepts or ideas. Each UMLS Metathesaurus concept is assigned at least one semantic type, and in some cases, several. In all cases, the most specific semantic type available in the network hierarchy is assigned to the concept.

The NCI Metathesaurus includes most of the UMLS Metathesaurus, with certain proprietary vocabularies of necessity excluded. In addition, the NCI Metathesaurus includes terminologies developed at NCI along with external vocabularies licensed by NCI. The local vocabularies developed at NCI are described in Table 2-1. As noted in the table, a limited model of the NCI Thesaurus is also accessible via the NCI Metathesaurus, as the NCI Source. Additional external vocabularies include MedDRA, SNOMED, ICD-O-3, and other proprietary vocabularies.

Vocabulary	*Content*	*Usage*
NCI Source	Limited model of the NCI Thesaurus	Reference terminology for cancer research applications
NCIPDQ	Expanded and re-organized PDQ	CancerLit indexing and clinical trials accrual
NCISEER	SEER terminology	Incidence reporting
CTEP	CTEP terminology	Clinical trials administration
MDBCAC	Topology and morphology	Cancer genome research
ELC2001	NCBI tissue taxonomy	Tissue classification for genetic data such as cDNA libraries
ICD03	Oncology	Cancer genome research and incidence reporting
MedDRA	Regulatory reporting terminology	Adverse event reporting
MMHCC	Mouse Cancer Database terminology	Mouse Models of Human Cancer Consortium
CTRM	Core anatomy, diagnosis, and agent terminology	Translational research by NCICB applications

Table 2-1 NCI local source vocabularies included in the Metathesaurus

Knowledge Representation and Description Logic

Knowledge representation has long been a prime focus in artificial intelligence research. This area of research asks how one can accurately encode the rich and highly detailed world of information that is required for the application area being modeled and yet, at the same time, capture the implicit common sense knowledge. One of the most common approaches to this problem in the 1970s was to utilize *frame-based representations*.

The basic idea of a frame is that important objects in our world fall into natural classes, and that all members of these classes share certain properties or attributes, called *slots*. For example, all dogs have four legs, a tail (or vestige of one), whiskers, etc. Restaurants generally have tables, chairs, eating utensils, and menus. Thus, when we enter a new restaurant or encounter a new dog, we already have a "frame of reference" and some expectation about the properties and behaviors of these entities.

In a seminal paper by Marvin Minsky, he placed the frame representation paradigm in the context of a semantic network of nodes, attributes, and relations. Figure 2-1 shows a simple frame-based representation of an earthquake, as it might be used in a semantic network of news stories. [1]

1. This example is excerpted from Artificial Intelligence, by Patrick Winston, Addison-Wesley, 1984.

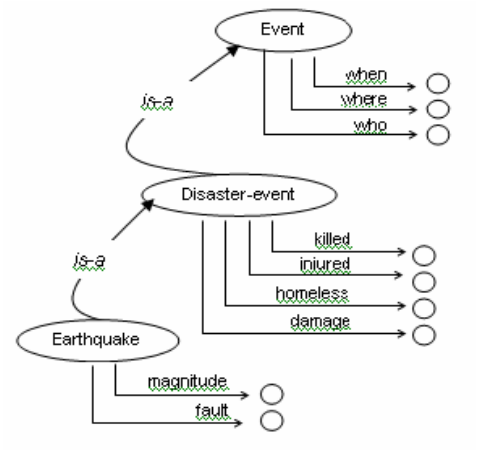

Figure 2-1 An earthquake in a semantic network of news stories

At the same time that frame-based representations were being explored, a popular alternative approach was to use (some subset of) first-order predicate logic (FOL), often implemented as a Prolog program. While propositional logic allows one to make simple statements about concrete entities, a complete first-order logic allows one to make general statements about anonymous elements with the introduction of variables as place-holders. The following example contrasts the difference in expressivity between propositional logic and FOL:

Propositional Logic	***First-order Predicate Logic***
All men are mortal. Socrates is a man. Socrates is mortal.	$\forall x : Man(x) \rightarrow Mortal(x)$ $Man(Socrates)$

In other words, in FOL it is possible to express general rules of inference that can be applied to all entities whose attributes satisfy the left-hand side of the inference operator. Thus, simply asserting *Man* (Socrates) entails *Mortal* (Socrates).

Since logic programming is based on the tenets of classical logic and comes equipped with automated theorem-proving mechanisms, this approach allowed the development of inference systems whose soundness and completeness could be rigorously demonstrated. However, while many of these early inference systems were logically sound and complete, they were often not very useful, as they could only be applied to highly proscribed areas or "toy problems". The problem was that a complete first-order predicate logic is itself computationally intractable, as certain statements may prove not to be decidable.

Suppose for example that we are trying to establish that some theorem, *P(x)*, is true. The way a theorem prover works is to first negate the theorem and, subsequently, to combine the negated theorem $(\neg P(x))$ with stored axioms in the body of knowledge to show that this leads to a logical contradiction. Ultimately, when the theorem prover derives the conclusion $P(x) \wedge \neg P(x)$, the program terminates and the theorem is considered proven.

This method of proof by refutation is guaranteed to terminate when it is indeed upheld by the body of knowledge. The problems arise when the initial theorem is not valid, as its negation may not produce a logical contradiction, and thus the program may not terminate.

In contrast, the frame representations offered a rich, intuitive means of expressing domain knowledge, yet they lacked the inference mechanisms and rigor that predicate logic systems could provide. As suggested by Figure 2-1, the frame representation captures a good deal of implicit knowledge. For example, we expect that all disaster events, including earthquakes, have information about fatalities and injuries and the extent of loss and property damage. In addition, we expect that these events will have locations, dates, and individuals associated with them.

Early efforts to apply predicate logic to frame representations in order to make this information explicit however, soon revealed that the problem was computationally intractable. This occurred for two reasons: (1) The frame representation was too permissive; more rigorous definitions were required to make the representation computational; and (2) the intractability of first-order predicate logic itself.

Several subsets of complete FOL have since been defined and successfully applied to develop useful computational models capable of significant reasoning. For example, the Prolog programming language is based on a subset of FOL that severely limits the use of negation. The family of description logic (DL) systems is a more recent development, and one that is especially well suited to the development of ontologies, taxonomies, and controlled vocabularies, as an important function of a DL is as an auto-classifier.

Description Logic

Description logic can be viewed as a combination of the frame-based approach with FOL. In the process, both models had to be scaled back to achieve an effective solution. Like frames, the DL representation allows for concepts and relationships among concepts, including simple taxonomic relations as well as other meaningful types of association. Certain restrictions however, are placed on these relations. In particular, any relation that involves class membership, such as the *isa* or *inverse-isa* relations, must be strictly acyclic.

The predicate logic used in a description logic system is also limited in various ways, depending on the implementation. For example, the minimal form of a DL does not allow any form of existential quantification. This limitation allows for a very easily computed solution space, but the resulting expressivity is severely diminished. The next step up in representational power allows limited existential quantification without atomic negation.

Indeed, today there is a large family of description logics that have been realized, with varying levels of expressivity and resulting computational complexities. In general, DLs are decidable subsets of FOL, and the decidability is due in large part to their acyclicity. The theory behind these models is beyond the scope of this discussion, and the interested reader is referred to *The Description Logic Handbook*, by Franz Baader, et al. (eds.), Cambridge University Press, 1993, ISBN number 0-521-78176-0.

The two main ingredients of a DL representation are *concepts* and *roles*. A major distinction between description logics and other subsets of FOL is its emphasis on set notations. Thus a DL concept never corresponds to a particular entity but rather to a *set* of entities, and the notations used for logical conjunction and disjunction are set intersection and union.

DL concepts can also be thought of as unary predicates in FOL. Thus the DL expression *Person* ∩ *Young* can be interpreted as the set of all children, with the corresponding FOL expression *Person(x)* /\ *Young(x)*. Syntactically then, DL expressions are variable free, with the understanding that the concepts always reference sets of elements.

A DL *role* is used to indicate a relationship between the two sets of elements referenced by

a pair of concepts. In general, DL notations are rather terse, and the concept (or set of elements) of interest is not explicitly represented. Thus, to represent the set of individuals whose children are all female, we would use \/x *hasChild.Female*. The equivalent expression in FOL might be something like:

\/x :hasChild(y.x) → female(x)

In terms of set theory, a role potentially defines the Cartesian product of the two sets. Roles can have restrictions, however, which place limitations on the possible relations. A *value* restriction limits the type of elements that can participate in the relation; a *number* restriction limits the number of such relations in which an element can participate.

In addition, each role defines a *directed* relation. For example, if *x* is the child of *y*, *y* is not also the child of *x*. In the above example *hasChild*, the parent concept is considered the *domain* of the relation, and the child is considered the *range*. Elements belonging to the set of objects defined by the range concept are also called role fillers. Number restrictions apply to the number of role fillers that are required or allowed in a relation. For example, a parent can be defined as a person having at least one child:

Person ∩ *(child)*

A DL representation is constructed from a ground set of *atomic concepts* and *atomic roles*, which are simply asserted. *Defined concepts* and *defined roles* are then derived from these atomic elements, using the set operations of intersection, union, negation, etc. Most DLs also allow existential and universal quantifiers, as in the above examples. Note, however, that these quantifiers always apply to the role fillers only.

The fundamental inference operation in DL is *subsumption*, and is usually indicated with subset notation. Concept A is said to subsume B, or $A \subseteq B$ when all members of concept B are contained in the set of elements defined by concept A, but not vice versa. That is, if B is a proper subset of A, then A subsumes B. This capability has far-reaching repercussions for vocabulary and ontology developers, as it enables the system to automatically classify newly introduced concepts. Moreover, correct subsumption inferencing can be highly nontrivial, as this generally requires examining all of the relationships defined in the system and the concepts that participate in those relations.

Description Logic in the NCI Thesaurus

The NCI Thesaurus is currently developed using the proprietary Apelon Inc. Ontylog™ implementation of description logic. Ontylog is distributed as a suite of tools for terminology development, management, and publishing. Although the underlying inference engine of Ontylog is not exposed, the implementation has the characteristics of what is called an AL- (Attributive Language) or FL- (Frame Language) description logic. It does not support atomic negation but does appear to provide all other basic description logic functionality.

The NCI Thesaurus is currently edited and maintained in the Terminology Development Environment (TDE) provided by Apelon. The TDE is an XML-based system that implements the DL model of description logic based on Apelon's Ontylog Data Model. The Data Model uses four basic components: *Concepts*, *Kinds*, *Properties*, and *Roles*. Use of the Apelon TDE for editing and maintenance if the NCI Thesaurus will change with the BioMedGT Wiki and Protégé 1.2 Tool Releases expected in early 2008.

As in other DL systems, *Concepts* correspond to nodes in an acyclic graph, and *Roles* correspond to directed edges defining relations between concept members. Each *Concept*

has a unique *Kind*. Formally, *Kinds* are disjoint sets of *Concepts* and represent major subdivisions in the NCI Thesaurus.

More concretely, *Kinds* are used in the *Role* definitions to constrain the *domain* and range values for that *Role*. Each *Role* is a *directed* relation that defines a triplet consisting of two concepts and the way in which they are related. The domain defines the *Concept* to which the *Role* applies, and the *Range* defines the possible values; in other words, *Concepts* that can fill that *Role*. For example, the Role *geneEncodes* might have its domain restricted to the *Gene_Kind* and its range to the *Protein_Kind*. This Role then essentially states that *Genes* encode *Proteins*.

As in all DLs, all roles are passed from parent to child in the inheritance hierarchy. For example, a "Malignant Breast Neoplasm" has the role *located-in*, connecting it to the concept "Breast". Thus, since the concept "Breast Ductal Carcinoma" *is-a* " Malignant Breast Neoplasm", it inherits the *located_in* relation to the "Breast" concept. These lateral nonhierarchical relations among concepts are referred to as associative or semantic roles; in contrast to the hierarchical relations that reflect the *is-a* roles. In the first-order algebra upon which Ontylog DL is based, every defined relationship also has a defined inverse relationship. For example, if *A* is contained by *B*, then *B* contains *A*. Inverse relationships are useful and are expected by human users of ontologies. However, they have a computational cost. If the edges connecting concept nodes are bi-directional, then the computation quickly becomes intractable. Therefore in the Ontylog implementation of DL, inverse relationships are not stored explicitly but computed on demand.

Concept Edit History in the NCI Thesaurus

One of the primary uses of the NCI Thesaurus is as a resource for defining tags or retrieval keys for the curation of information artifacts in various NCI repositories. However, since these tags are defined at a fixed point in time, they necessarily reflect the content and structure of the NCI Thesaurus at that time only. Given the rapidly evolving terminologies associated with cancer research, there is no guarantee that the tags used at the time of curation in the repository will still have the same definition in subsequent releases of the Thesaurus. In most cases the deprecation or redefinition of a previously defined tag is not disastrous, but it may compromise the completeness of the information that can be retrieved.

In order to address this issue, the EVS team has developed a *history* mechanism for tracing the evolution of concepts as they are created, merged, modified, split, or retired. (In the NCI Thesaurus, no concept is ever deleted.) The basic idea is that each time an edit action is performed on a concept, a record is added to a history table. This record contains information about relations that held for that concept at the time of the action as well as other information, such as version number and timestamp that can be use to reconstruct the state when the action was taken (Table 2-2).

Column Name	***Description***
History_ID	Unique consecutive number for use as the database primary key
Concept_Code	The concept code for the concept currently being edited
Action	Edit Action: {Create, Modify, Split, Merge, Retire}
Baseline_Date	Date of NCI Thesaurus Baseline (see discussion below)
Reference_Code	This field contains the concept code of a second concept either participating in or affected by the editor's action. Captures critical

Column Name	Description
	information concerning the impact of the edit actions on other concepts. The value will always be null if the action is Create or Modify.

Table 2-2 Summary of the information stored in the history table

Capturing the history data for a Split, Merge, or Retire action is more complicated. In a Split, a concept is redefined by partitioning its defining attributes between two concepts, one of which retains the original concept's code and one that is newly created. This action is taken when ambiguities in the original concept's meaning require clarification by narrowing its definition.

In the case of a Split, three history records will be created: one for the newly created concept, (with a null Reference_Code), and two for the original concept that is being split. In the first of these two records, the Reference_Code is the code for the new concept; in the second it is the code of the split concept (Figure 2-2).

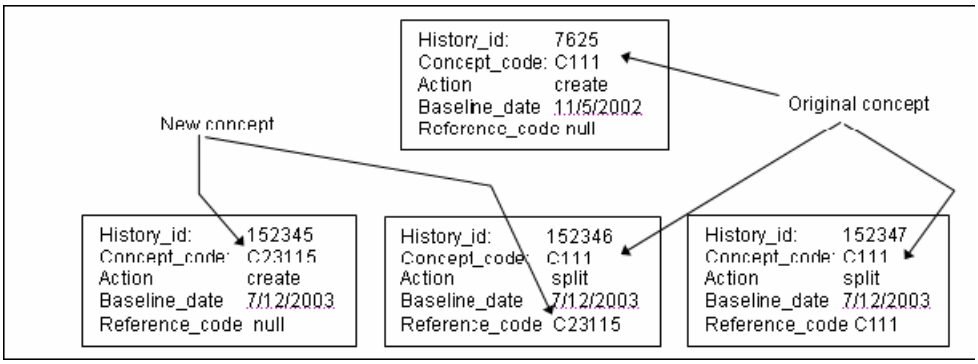

Figure 2-2 History records for the split action

For Merge actions, the situation is similar to a Split. In this case, two ambiguous concepts must be combined, and only one of the original concepts is retained. Again there will be three history records created: two for the concept that will be retired during the merge, and one for the "winning" concept. The Reference_Code in the history record for the "winning" concept will be the same as the Concept_Code; that is, the concept points to itself as a descendant in the Merge action. The Reference_Code will be null in one of the entries for the retiring concept, while the second entry will have the code of the "winning" concept; thus, this Reference column points to the concept into which the concept in the Concept_Code column is being merged.

Finally, if the action is Retire, there will be as many history entries as the concept has parent concepts. The Reference column in these entries will contain the concept code of the parent concepts, one parent concept per history entry. The motivation for this is that end-users with documents coded by such retired concepts may find a suitable replacement among the concept's parents at the time of retirement.

The caCORE EVS APIs support concept history queries.

Downloading the NCI Thesaurus

The NCI Thesaurus can be downloaded in a couple of formats, including simple tab-delimited ASCII format and OWL format (the Web Ontology Language). The ASCII-

formatted files are available for download at the NCICB download site, as ThesaurusV2_0Flat.zip and ThesaurusV2_0XML.zip. The OWL formatted version is available at http://ncicb.nci.nih.gov/xml/owl/EVS/Thesaurus.owl. Users who prefer to use FTP for download can go to the caCORE FTP site. The format of the ASCII flat file is extremely simple. For each concept, the download file includes the following information:

1. The concept code: all terms have the "C" prefix, followed by its integer index;
2. The concept name: this name may contain embedded punctuation and spaces;
3. A pipe-delimited list of parent concepts, as identified in the NCI Thesaurus by *isa* relations;
4. A pipe-delimited list of synonyms, the first of which is the preferred name; and
5. One of the NCI definitions for the term-if one exists. Each of these separate types of information is tab-delimited; within a given category, the individual entries are separated by pipes ("|"). Only the third and fourth categories, i.e., the parent concepts and synonyms, have multiple entries requiring the pipe separators. Note that while much of the information available from the interactive Meta-phrase server is included in the download, any information outside the NCI Thesaurus description logic vocabulary (e.g., Diagnosis, Laboratory, Procedures, etc.) is not.

For example, the flat file download for the term "*Mercaptopurine*" is as follows:

```
C6 Mercaptopurine Immunosuppressants|Purine Antagonists

Mercaptopurine|1,3-AZP|1,7-Dihydro-6H-purine-6-thione|3H-

Purine-6-thiol|6

Thiohypoxanthine|6 Thiopurine|6-MP|6-Mercaptopurine|6-

Mercaptopurine

Monohydrate|6-Purinethiol|6-Thiopurine|6-Thioxopurine|6H-

Purine-6-thione,

1,7-dihydro- (9CI)|6MP|7-Mercapto-1,3,4,6-

tetrazaindene|AZA|Alti-

Mercaptopurine|Azathiopurine|BW 57-323H|CAS

50442|Flocofil|Ismipur|Leukerin|Leupurin|MP|Mercaleukim|Mercaleukin|Merc
ap|Mercaptina|Mercapto-6-purine|Mercaptopurinum|Mercapurin|Mern|NCI-
C04886|NSC755|Puri-Nethol|Purimethol|Purine-6-thiol (8CI)|Purine-6-thiol

Monohydrate|Purine-6-thiol, Monohydrate|Purinethiol|Purinethol|U-
4748|WR-2785
An anticancer drug that belongs to the family of drugs called
antimetabolites.
```

Users who wish to use an encoded format rather than the simple ASCII form should download the OWL encoding of the NCI Thesaurus, which is described below.

OWL Encoding of the NCI Thesaurus

OWL, as specified and proposed by the World Wide Web Consortium (W3C), is an emerging standard for the representation of semantic content on the web. Building on the earlier groundwork laid by XML, the Resource Description Framework (RDF) and RDF schema; and subsequently, by DAML+OIL, OWL represents the culmination of what has been learned from these previous efforts.

While XML provides surface syntax rules and XML Schema provides methods for validating a document's structure, neither of these can in itself impose semantic constraints on how a document is interpreted. RDF provides a data model for specifying objects (resources) and their relations, and RDF Schema allows one to associate properties with the individual resources as well as taxonomic relations among the objects. Yet even these extensions could not provide the breadth and depth of representation needed to encode nontrivial real-world information. OWL adds vocabulary for describing arbitrary nonhierarchical relations between classes, cardinality constraints, resource equivalences, richer typing of properties, and enumerated classes.

A major focus of the W3C is the establishment of the Semantic Web, which is a far-reaching infrastructure whose purpose is to provide a framework whereby autonomous self-documenting agents and web services can exchange meaningful information without human intervention. OWL is the first step towards realizing this vision. Because of collaborative efforts with Dr. James Hendler and the University of Maryland, the NCI Thesaurus is now available for download in OWL format; this section describes the mapping of the NCI Thesaurus to OWL. The mapping of the NCI Thesaurus into OWL format proceeds via the Ontylog XML elements declared in Apelon's Ontylog DTD. The four basic elements are *Kinds*, *Concepts*, *Roles*, and *Properties*, where:

- Kinds are the top-level super classes in the Thesaurus; they enumerate the different possible categories of all concepts, and include such things as Anatomy, Biological Processes, Chemicals and Drugs, etc. *Each NCI Thesaurus Kind is converted to an owl:Class*.
- An NCI Thesaurus Concept describes a specific concept under one of the Kind categories. *Each NCI Thesaurus Concept is converted to an owl:Class*.
- Roles capture how concepts relate to one another. Generally, Roles have restricted domains and ranges, which limit the sets of concepts that can participate in the Role according to their categories, for example Kinds. The "defining roles" within a concept definition provide these local restrictions on the ranges of roles. *Each NCI Thesaurus Role is converted to an owl:ObjectProperty*.
- NCI Thesaurus Properties encode the attributes that pertain to a class; they contain metadata that describes the class, but not its instantiations or subclasses. *Each NCI Thesaurus Property is converted to an owl:AnnotationProperty*.

The bulk of the Thesaurus comprises concept definitions; this is also where the most complex semantics occur. Each concept in the Thesaurus has three main types of associated data: defining concepts, defining roles, and properties. A "defining concept" is essentially a super class; the defined concept in OWL has an *rdfs:subClassOf* relationship to the defining concept.

The defining roles and properties are mapped as described above; the *owl:Annotation-Property* is actually a subclass of *rdf:Property*, and, like *rdfs:comment* and *rdfs:label*, can be attached to any class, property or instance. This allows properties from the Thesaurus to be associated directly with a concept's corresponding class, without violating the rules of OWL.

In addition to any explicitly named properties, each element in the Thesaurus also has a uniquely defined "code" and "id" attribute associated with it. These are used as unique identifiers in the Apelon development software, and, as such, are not defined explicitly as roles or properties. In mapping these identifying attributes to OWL, we have treated these as special cases of the explicit property elements. Just like other properties in the Thesaurus, they are mapped as owl:AnnotationProperties. Table 2-3 summarizes the mapping of elements in the Ontylog DTD to OWL elements.

Ontylog Name Conversion

In mapping to OWL, all Ontylog concept *names* must be converted to proper RDF identifiers (rdf:id) following the RDF naming rules. This is achieved by removing any spaces in the original names and substituting all illegal characters with underscores. Names that begin with numbers are also prefixed with underscores to make them legal. The original concept name however, is preserved as an rdfs:label. The following steps summarize the conversion of names:

1. Any "+" characters are replaced with the text "plus".
2. All role names are prefixed with an "r" to ensure that roles and properties with the same name do not clash.
3. Any characters that are not alphanumeric, or one of "-" and "_," are replaced with an underscore ("_").
4. All names with leading digits are prefixed with an underscore.
5. Multiple adjacent underscores in the corrected name are replaced with a single underscore.

Ontylog Element	***Owl Element***	***Comment***
kindDef	owl:Class	
roleDef	owl:ObjectProperty	
propertyDef	owl:AnnotationProperty	
conceptDef	owl:Class	
name*	rdf:ID	Applies to the name subelement of *kindDef*, *roleDef*, *propertyDef*, and
name	rdfs:label	Because the *conceptDef name* contains some useful semantics, the original form is retained as an *rdfs:label*. No other name elements are retained in *rdfs:label*.

Ontylog Element	Owl Element	Comment
Code	owl:AnnotationProperty	Defined as an *owl:AnnotationProperty* with *rdf:ID*="code". Code values remain the same for each concept.
Id	owl:AnnotationProperty	Defined as an *owl:AnnotationProperty* with *rdf:ID*="ID". ID values remain the same for each concept.
definingConcepts	rdfs:subClassOf	The *concept* subelement of *definingConcepts* is mapped to the *rdf:resource* attribute of the *rdfs:subClassOf* element.
Domain	rdfs:domain	
Range	rdfs:range	
definingRoles / role / name	owl:onProperty	*definingRoles* are converted to owl restrictions on properties. The *name* child element of *definingRoles/role* is taken as the *rdf:resource* attribute of the *owl:onProperty* element.
definingRoles / role / value	owl:someValuesFrom	*definingRoles* are converted to owl restrictions on properties. The *value* child element of *definingRoles/role* is taken as the *rdf:resource* attribute of the *owl:someValuesFrom* element.

Table 2-3 Ontylog DTD to OWL Conversions

Note: Name Ontology elements are converted to rdf:ID as described in the Ontylog Name Conversion section. namespaceDef and namespace elements are not mapped to OWL.

Additional information about the Ontylog encoding is available in the Ontylog DTD, which can be downloaded from the NCICB EVS FTP site, along with the zipped ASCII flat file and the Ontylog XML encoding. The current OWL translation of the NCI Thesaurus contains over 500,000 triples and is available in zipped format from the FTP site, as well as in unzipped format at http://ncicb.nci.nih.gov/xml/owl/EVS/Thesaurus.owl, the mindswap web site for download or online viewing.

Ontylog Mappings

Mapping of Gene Ontology to Ontylog

The LexBIG Terminology Server provides access to the Gene Ontology™ Consortium's (GO) controlled vocabulary. The GO ontologies are widely used-most likely due to their simplicity of design and their potential for automated transfer of biological annotations, from model organisms to more complex organisms based on sequence similarities. GO

comprises three independent controlled vocabularies (ontologies) encoding biological process, molecular function, and cellular components for eukaryotic genes. GO terms are connected via two relations, *is-a* and *part-of*, that define a directed acyclic graph. Although concepts in the ontologies were initially derived from only three model systems (yeast, worm, and fruit fly), the goal was to encode concepts in such a way that the information is applicable to *all* eukaryotic cells. Thus, species-specific anatomies are not represented, as this would not support a unifying reference for species-divergent nomenclatures.

Each month NCI will load the latest version of GO into a test instance of the DTS server, and, following validation in the Ontylog environment, will promote it to a production server for programmatic access by NCI applications. NCI converts GO into the Ontylog XML representation (necessary for import into the DTS server) via a stylesheet transformation followed by some post-processing to satisfy Ontylog constraints. It is NCI's intent that the version of GO on the DTS server will not be more than a month behind the current version available from http://www.geneontology.org. However, it might be necessary to skip releases if unforeseen complications arise.

Table 2-4 and Table 2-5 summarize the encoding of GO elements into Ontylog.

Ontology Element	***Instance Name (and optional description)***
namespaceDef	**GO**
kindDef	**GO_Kind**
RoleDef	**part-of** This role is unused; however, the software requires that at least one role be declared.
propertyDef	**Preferred_Name**
propertyDef	**Synonym**
propertyDef	**DEFINITION**
propertyDef	**Dbxref** complex property containing two XML-marked up GO entities: "go:database_symbol," and "go:reference," using tags "database_symbol" and "reference," respectively.
propertyDef	**part-of** complex property containing two XML-marked up GO entities: "go:name" and "go:accession," using tags "go-term" and "go-id," respectively.

Table 2-4 Ontylog elements used for GO mapping

The go:name stored in Preferred_Name is as declared in GO. However, the go:name used in the Ontylog name might have been modified during the conversion process (by appending underscores) to make the Ontylog name unique.

GO term element	conceptDef element	(propertyDef)
go:accession	Code	
go:name	Name	
go:isa	definingConcepts	
go:name	Property	Preferred_Name
go:synonym	Property	Synonym
go:definition	Property	DEFINITION
go:part-of	Property	part-of
go:dbxref	Property	dbxref

Table 2-5 Mapping of GO term to Ontylog conceptDef

Mapping of MedDRA to Ontylog

Vocabulary Hierarchy Structure

The Ontylog version of MedDRA reflects the native hierarchy, with terms organized according to their term type as shown in Figure 2-3.

SOC (System Organ Class)

|__ HLGT (High Level Group Term)

|__ HLT (High Level Term)

|__ PT (Preferred Term)

|__LLT (Lowest Level Term)

Figure 2-3 Hierarchy of MedDRA

The Special Search Categories (SSC) are under the concept *AssociativeTerm-Group(SSCs)*, which has been created by the EVS as a header concept for the SSC terms to be grouped together. All the System Organ Class (SOC) concepts as well as the top header concept for the SSCs are under the *MedDRA[V-MDR]* root node. Although Low Level Terms (LLTs) can have any type of relationship to their Preferred Term (PT) (for example, a synonym of the PT), the Ontylog version presents them all as children concepts. The *Associative Term Group (SSCs)* concept has a special code and term type not found in MedDRA to distinguish it from other terms in the vocabulary.

Concept Codes and Names

The concept name is created from the MedDRA term followed by the MedDRA code enclosed in brackets. The Ontylog concept name must be unique so including the code in

the name guarantees uniqueness. For display purposes, the property `Preferred_Name` should be used instead of the concept name; it contains the unadorned MedDRA term. The Ontylog concept code is the MedDRA code.

Roles

A single role has been defined. The role `has_associated_term` is utilized to relate SSC top-level categories with their associated PT terms. All the concepts in the vocabulary are primitive.

Properties

The properties defined for MedDRA 6 in Ontylog are shown in Table 2-6; their provenance in the MedDRA distribution is indicated.

Ontology Property	***MedDRA Entity***
Code_in_Source	MedDRA code (llt_code, pt_code, hlt_code, and so forth)
Cross-reference, cross reference to WHOART	COSTART, ICD9-CM, and so forth
Descriptor_ID	pt_code of an LLT
MedDRA_Abbreviation	soc_abbrev, spec_abbrev
NCI_META_CUI	-
Preferred_Name	MedDRA name (llt_name, pt_name, hlt_name, and so forth)
Primary_SOC	pt_soc_code of a PT
Serial_Code_International_SOC_Sort_Order	intl_ord_code
Term_Type	-
UMLS_CUI	-

Table 2-6 Properties defined in the Ontylog version of MedDRA. Properties that are not derived directly from MedDRA have a dash in the MedDRA Entity column.

Of the MedDRA-derived properties, only Cross-reference is not a straightforward name-value pair. This property has subfields encoded in xml; the xml elements are source and source code, where the source code contains a code or symbol assigned by an external vocabulary source to a specific term.

Two properties are not derived from MedDRA: NCI_META_CUI and UMLS_CUI. These properties contain the Concept Unique Identifier (CUI) of concepts in the NCI Metathesaurus containing MedDRA terms. The property name indicates whether the CUI is assigned to the concept by Unified Medical Language System (UMLS) or by NCI. The Term_Type property is indirectly derived from MedDRA and indicates the hierarchy level of a term with the term types as shown in Figure 2-3; in addition, the term type for Obsolete Lower Level Terms (OLLT) is also used.

Mapping of MGED Ontology to Ontylog

The native MGED Ontology (MO) is edited in OilEd and distributed in the Defense Advanced

Research Projects Agency (DARPA) Agent Markup Language (DAML) + Ontology Inference Layer (OIL) XML format. DAML+OIL can be converted to the Ontylog Description Logic (DL) in a relatively straightforward manner. However, some valid DAML+OIL constructions cannot be represented in Ontylog DL, including enumerations and specific combinations of ObjectProperties that result in classification cycles in Ontylog. In MO version 1.1.9, two ObjectProperties have been asserted near the top of the hierarchy on the MGEDCoreOntology class. On conversion to Ontylog these assertions generate classification cycles, however, the data cannot be massaged as was done in preliminary conversions with previous versions of MO because the fix would have required modifications to every converted concept. Consequently, beginning with MO v 1.1.9, all the ObjectProperties in DAML+OIL are converted to Ontylog properties (rather than Ontylog roles), which are annotations ignored by the classifier.

Vocabulary Hierarchy Structure

The MO class hierarchy structure is preserved in the Ontylog conversion. One minor difference is that MO class instances are also represented in the Ontylog concept hierarchy (since there is no distinction between classes and instances in Ontylog). A non-MO top-level concept, OrphanConcepts, has been added in the Ontylog representation to hold MO instances of Thing.

Concept IDs, Codes, and Names

MO classes and instances are identified solely by their name; no codes or numeric IDs are assigned. For the conversion to Ontylog, MO class or instance names are retained as concept names. As Ontylog concepts also require unique codes and IDs, a code and an ID are created during the conversion. The ID reflects the position of the class or instance in the XML tree. The code is derived from the ID by adding an “X-MO-” prefix to it; therefore, the code is not guaranteed to remain invariant from version to version of the MO. A mapping table is made available whenever the MO is updated.

Roles

No roles have been defined; all the concepts are primitive.

Properties

All the object and datatype properties defined in MO have been converted to Ontylog properties. With the exception of has_reason_for_deprecation and has_database, all the properties have been 'manually” propagated to children concepts in the database in order to mimic the expected role inheritance. In addition, new properties have been defined as shown in Table 2-7.

Ontylog Property	***MGED Ontology Entity***
DEFINITION	rdfs:comment value
Preferred_Name	rdf:about value
Synonym	rdf:about value
Concept_Type	-

Table 2-7 New properties defined in the Ontylog version of the MGED Ontology and their provenance (if applicable) in the daml+oit file

The `Preferred_Name` property is recommended for display purposes, while `Synonym` is recommended for searches by dependent applications (even though the value of both properties is the same, the EVS tries to maintain a certain consistency in the usage of properties for the benefit of all users). The `Concept_Type` property holds one of two values: `mged_class`, or `mged_instance`.

LexBIG

caCORE EVS is the adopter site for the open source public domain terminology server LexBIG, developed by the Mayo Clinic as part of the caBIG Program. The goal of caCORE EVS is to adopt LexBIG as the sole terminology server infrastructure for EVS. The Apelon DTS server is a proprietary server that does not allow exposure of the API. As a result, caCORE EVS 3.2 and earlier have provided a custom API that communicates directly with the DTS Server and is publicly available. The caCORE EVS 4.0 release begins the transition to LexBIG by re-exposing the caCORE EVS 3.2 custom API with LexBIG as the backend terminology server. Additionally the caCORE EVS 3.2 API will continue to be supported for approximately one year. This gives users who have implemented to the caCORE EVS 3.2 API time to plan for the removal of the Apelon DTS server.

LexBIG is based on the LexGrid Model, Mayo's proposal for standard storage of controlled vocabularies and ontologies. The LexGrid Model defines how vocabularies should be formatted and represented programmatically, and is intended to be flexible enough to represent accurately a wide variety of vocabularies and other lexically based resources. The model also defines several different server storage mechanisms (for example, relational database, LDAP) and a XML format. This model provides the core representation for all data managed and retrieved through the LexBIG system, and is now rich enough to represent vocabularies provided in numerous source formats including: Open Biomedical Ontologies (OBO) , Web Ontology Language (OWL), and the Unified Medical Language System (UMLS) Rich Release Format (RRF). This common model is a critical component of the LexGrid project. Once disparate vocabulary information can be represented in a standardized model, it becomes possible to build common repositories to store vocabulary content and common programming interfaces and tools to access and manipulate that content.

LexBIG has three major components: Service Management tools to load, index, and manage vocabulary content for the vocabulary server; an API providing Java interfaces to various functions including lexical queries, graph representation and traversal, and NCI change event history; and a Graphical User Interface providing access to service management and API functions. The LexBIG API enables querying information stored in the LexGrid model. Similar APIs have been developed for LexBio that are used at the National

Center for Bioontologies (NCBO). NCI EVS has adopted and modified the NCBO's BioPortal as a web browser for LexBIG. The NCI BioPortal can be accessed at http://bioportal.nci.nih.gov.

In summary LexBIG provides the following features:

- A robust and scalable open source implementation of EVS-compliant vocabulary services. The API specification will be based on but not limited to fulfillment of the caCORE EVS API. The specification will be further refined to accommodate changes and requirements based on prioritized needs of the caBIG™ community.
- A flexible implementation for vocabulary storage and persistence, allowing for alternative mechanisms without affecting client applications or end users. Initial development will focus on delivery of open source freely available solutions, though this does not preclude the ability to introduce commercial solutions (for example, Oracle).
- A standard tool for loading and distribution of vocabulary content. This includes but is not limited to support of standardized representations such as UMLS Rich Release Format (RRF), the OWL web ontology language, and Open Biomedical Ontologies (OBO).

Chapter 3 Overview to caCORE

This chapter provides an overview of the NCI CBIIT caCORE infrastructure.

Topics in this chapter include:

Architecture Overview

The NCI Center for Bioinformatics (NCICB) provides biomedical informatics support and integration capabilities to the cancer research community. NCICB has created a core infrastructure called Cancer Common Ontologic Representation Environment (caCORE), a data management framework designed for researchers who need to be able to navigate through a large number of data sources. By providing a common data management framework, caCORE helps streamline the informatics development throughout academic, government and private research labs and clinics. The components of caCORE support the semantic consistency, clarity, and comparability of biomedical research data and information. caCORE is open-source enterprise architecture for NCI-supported research information systems, built using formal techniques from the software engineering and computer science communities. The four characteristics of caCORE include:

- Model Driven Architecture (MDA)
- n-tier architecture with open Application Programming Interfaces (APIs)
- Use of controlled vocabularies, wherever possible
- Registered metadata

The use of MDA and *n*-tier architecture, both standard software engineering practices, allows for easy access to data, particularly by other applications. The use of controlled vocabularies and registered metadata, less common in conventional software practices, requires specialized tools, generally unavailable.

As a result, the NCI CBIIT (in cooperation with the NCI Office of Communications) has developed the Enterprise Vocabulary Services (EVS) system to supply controlled vocabularies, and the Cancer Data Standards Repository (caDSR) to provide a dynamic metadata registry. When all four development principles are addressed, the resulting system has several desirable properties. Systems with these properties are said to be “caCORE-like”.

1. The *n*-tier architecture with its open APIs frees the end user (whether human or machine) from needing to understand the implementation details of the underlying data system to retrieve information.
2. The maintainer of the resource can move the data or change implementation details (Relational Database Management System, and so forth) without affecting the ability of remote systems to access the data.
3. Most importantly, the system is ‘semantically interoperable’; that is, there exists runtime-retrievable information that can provide an explicit definition and complete data characteristics for each object and attribute that can be supplied by the data system.

Components of caCORE

The components that comprise caCORE are EVS, caDSR, caBIO, CSM, and CLM. Each is described briefly below.

Enterprise Vocabulary Services (EVS)

EVS provides controlled vocabulary resources that support the life sciences domain, implemented in a description logics framework. EVS vocabularies provide the semantic 'raw material' from which data elements, classes, and objects are constructed.

Cancer Data Standards Repository (caDSR)

The caDSR is a metadata registry, based upon the ISO/IEC 11179 standard, used to register the descriptive information needed to render cancer research data reusable and interoperable. The caBIO, EVS, and caDSR data classes are registered in the caDSR, as are the data elements on NCI-sponsored clinical trials case report forms.

Cancer Bioinformatics Infrastructure Objects (caBIO)

The caBIO model and architecture is the primary programmatic interface to caCORE. Each of the caBIO domain objects represents an entity found in biomedical research. Unified Modeling Language™ (UML) models of biomedical objects are implemented in Java as middleware connected to various cancer research databases to facilitate data integration and consistent representation. Examining the relationships between these objects can reveal biomedical knowledge that was previously buried in the various primary data sources.

Common Security Model (CSM)

CSM provides a flexible solution for application security and access control with three main functions:

- Authentication to validate and verify a user's credentials
- Authorization to grant or deny access to data, methods, and objects
- User Authorization Provisioning to allow an administrator to create and assign authorization roles and privileges.

Common Logging Module (CLM)

CLM provides a separate service under caCORE for Audit and Logging Capabilities. It also comes with a web based locator tool. It can be used by a client application directly, without the application using any other components like CSM.

In September of 2007, the NCI CBIIT Infrastructure and Product Management Team made the decision to separate the caCORE Components that had previously been bundled and released together. This decision was geared toward allowing each of the infrastructure product teams to be more responsive in addressing specific needs of the user community. caCORE EVS is the first component to release under the new release paradigm. With each component release there is a product specific Technical Guide.

This particular guide focuses on the EVS component of caCORE 4.0. For more details on the other components, refer to the caCORE Overview page at http://ncicb.nci.nih.gov:80/NCICB/infrastructure/cacore_overview, which directs you to other product specific Technical Guides.

Chapter 4 caCORE EVS Architecture

This chapter describes the architecture of the caCORE EVS. It includes information about the client-server communication. It also describes the layout of the system software packages.

Topics in this chapter include:

caCORE EVS System Architecture

The caCORE EVS infrastructure exhibits an n-tiered architecture with client interfaces, server components, backend objects, and additional backend systems (Figure 4-1). This n-tiered system divides tasks or requests among different servers and data stores. This isolates the client from the details of where and how data is retrieved from different the LexBIG terminology server.

Clients (browsers, applications) receive information from backend objects. Java applications also communicate with backend objects via domain objects packaged within the evs-client.jar. Non-Java applications can communicate via SOAP (Simple Object Access Protocol). Back-end objects communicate directly with the LexBIG API.

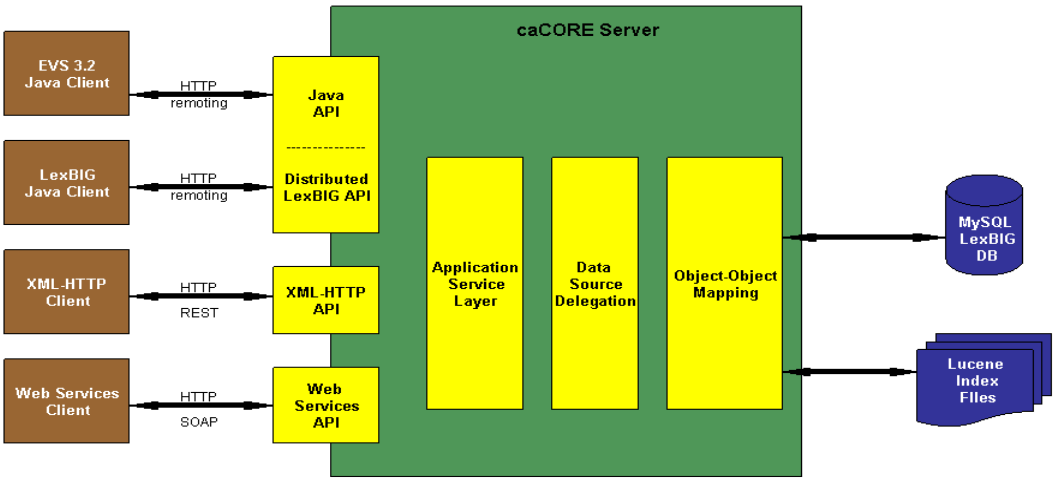

Figure 4-1 caCORE/EVS Architecture

Most of the caCORE EVS infrastructure is written in the Java programming language and leverages reusable, third-party components.

The infrastructure is composed of the following layers:

The Application Service layer — consolidates incoming requests from the various interfaces and translates them to native query requests that are then passed to the data layers. All interfaces provide full, anonymous read-only access to all data.)

The Data Source Delegation layer — is responsible for conveying each query that it receives to the respective LexBIG Service objects that can perform the query.

Object-Object Mapping (OOM) — is performed by objects that follow the façade design pattern. These objects make the task of accessing a large number of modules/functions much simpler by providing an additional interface layer that allows it to interact with the rest of the caCORE EVS system.

LexBIG Service API — The LexBIG Service is designed to run standalone or as part of a larger network of services. It is comprised of four primary subsystems: Service Management, Service Metadata, Query Operations, and Extensions. The Service Manager provides administration control for loading a vocabulary and activating a service. The Service Metadata provides external clients with information about the vocabulary content (that is, NCI Thesaurus) and appropriate licensing information. The Query Operations provide numerous functions for querying and traversing vocabulary content. Finally, the extensions component provides a mechanism to extend the specific service functions, such as Loaders, or re-wrap specific query operations into convenience methods. For more information refer to the LexBIG Programmer's Guide.

Client Technologies

Applications using the Java programming language can access EVS directly through the domain objects provided by the evsapi-client.jar (see Chapter 4) The network details of the communication to the caCORE EVS server are abstracted away from the developer. Hence developers need not deal with issues such as network and database communication, but can instead concentrate on the biological problem domain.

The caCORE EVS system also allows non-Java applications to use SOAP clients to interface with caCORE EVS Web services. SOAP is a lightweight XML-based protocol for the exchange of information in a decentralized, distributed environment. It consists of an envelope that describes the message and a framework for message transport. caCORE EVS uses the open source Apache Axis package to provide SOAP-based web services to users. This allows other languages, such as Python or Perl to communicate with caCORE EVS objects in a straightforward manner.

The caCORE EVS architecture includes a presentation layer that uses a J2SE application server (such as Tomcat or JBoss). The JSPs (Java Server Pages) are web pages with Java embedded in the HTML to incorporate dynamic content in the page. caCORE EVS also employs Java Servlets, which are server-side Java programs, that web servers can run to generate content in response to client requests. All logic implemented by the presentation layer uses Java Beans, which are reusable software components that work with Java. All caCORE EVS objects can be transformed into XML, the eXtensible Markup Language, as a universal format for structured data on the Web.

Communication between the client interfaces and the server components occurs over the Internet using the HTTP protocol. The server components are deployed in a web

application container as a .war (Web archive) file that communicates with the LexBIG terminology server.

caCORE EVS Software Packages

The caCORE EVS software is comprised of several java packages. A few of the significant packages include:

- gov.nih.nci.evs.domain
- gov.nih.nci.evs.query
- gov.nih.nci.lexbig

The caCORE EVS `domain` package (Figure 4-2) provides access to the Java 3.2 interfaces and classes such as DescLogicConcept, MetaThesaurusConcept, etc. For a complete list of domain objects, see the EVS 3.2 Object Model.

The `query` package contains classes that facilitate a custom query mechanism for EVS domain objects and is discussed in Chapter 5 Interacting with caCORE EVS.

The `lexbig` package contains the Distributed LexBIG (DLB) Adapter classes discussed in more detail later in this guide.

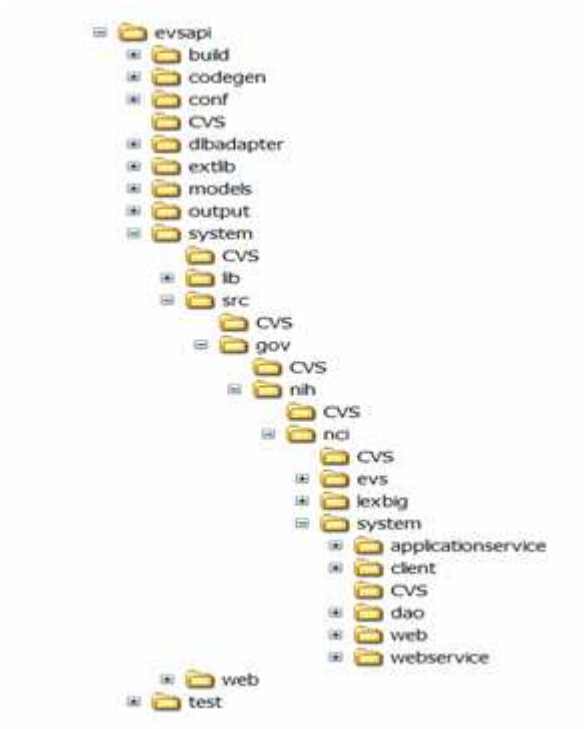

Figure 4-2 caCORE EVS packages

In addition to domain packages, the caCORE EVS API specification includes the framework packages described in the following subsections.

System

The system package contains several subpackages including a `lib` folder that holds the

third-party libraries required to deploy the system. It also contains the `src` folder that contains the bulk of the EVS system code. The `src` subpackages include the `EVSQuery` classes, the Distributed LexBIG Adapter (DLBAdapter) classes, and the `system` package. The `system` package contains the following categories of subpackages: application service package (described in the Client Technologies on page 28), data access package, delegate/service locator package, proxy package, and web service package.

- **EVS Query –** the gov.nih.nci.evs.query package contains the *evsQuery* and *evsQueryImpl* java interface and class.
- **DLBAdapter –** the gov.nih.nci.lexbig.ext package contains the *DLBAdapter classes.* These convenience methods are used to supplement access to the Distributed LexBIG API (described in Chapter 5).
- **Data Access –** The data access package (gov.nih.nci.system.dao) is the layer at which the query is parsed from objects to the native query, the query is executed, and the result sets are converted back to domain objects results. This layer has implementation for external data access layer for querying other subsystems. It also contains the security objects required to support the controlled access requirements to the MedDRA data source.
- **Proxy Interface –** The proxy interface package (gov.nih.nci.system.client.proxy) is the gateway for the requests from Java and platform independent web service clients.
- **Web Service -** The Web service package (gov.nih.nci.system.webservice) contains the Web service wrapper class that uses Apache's Axis.
- **Web -** The Web package (gov.nih.nci.system.web) contains the useful utilities.

Chapter 5 Interacting with caCORE EVS

This chapter describes the components of the caCORE EVS 4.0 release, the service interface layer provided by the EVS API architecture, and gives examples of how to use the EVS 4.0 APIs. This chapter also describes the Distributed LexBIG API and the Distributed LexBIG Adapter.

Topics in this chapter include:

caCORE EVS Components

The caCORE EVS API is a public domain open source wrapper that provides full access to the LexBIG Terminology Server. LexBIG hosts the NCI Thesaurus, the NCI Metathesaurus and several other vocabularies. Java clients accessing the NCI Thesaurus and Metathesaurus vocabularies communicate their requests via the open source caCORE EVS APIs as shown in Figure 5-1.

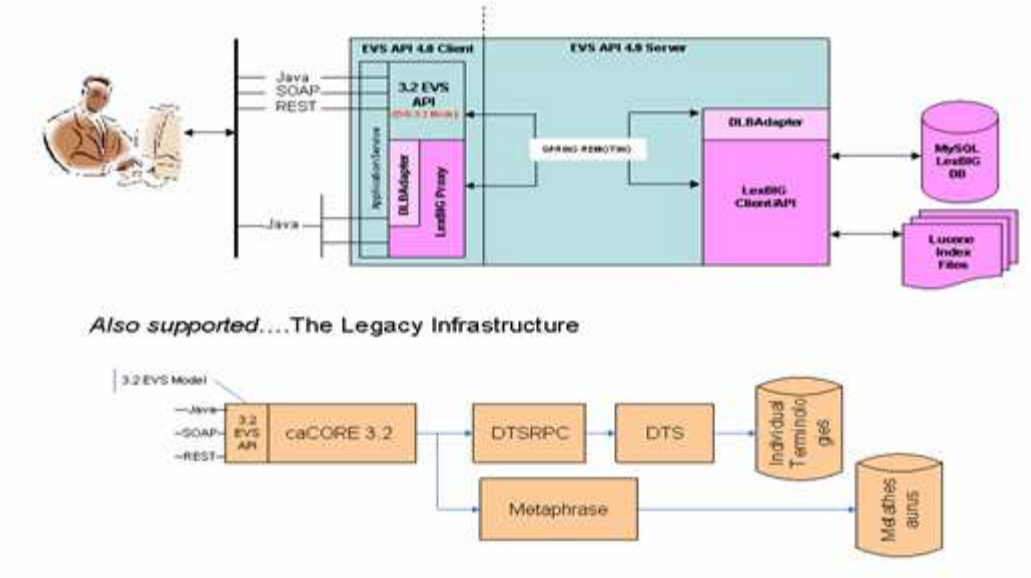

Figure 5-1 Overview of the caCORE EVS 4.0 Release Components

The open source interfaces provided as part of caCORE EVS 4.0 include Java APIs, a SOAP interface, and an HTTP REST interface. The Java APIs are based on the following object models:

- the EVS 3.2 object model
- the LexBIG Service object model

The EVS 3.2 model, exposed as part of caCORE 3.2, has been re-released with LexBIG as the backend terminology service versus the proprietary Apelon DTS backend. The SOAP and HTTP REST interfaces are also based on the 3.2 object model. The SDK 4.0 was used to generate the EVS 3.2 Java API, as well as the SOAP and HTTP REST interfaces.

Notes: The only difference between the EVS 3.2 API exposed as part of the caCORE EVS 4.0 and that exposed as part of caCORE 3.2 is the backend terminology server used to retrieve the vocabulary data. The interface (API calls) are the same and should only require minor adjustments to user applications. You are not able to integrate caCORE 3.2 components with caCORE EVS 4.0. If you used multiple components of caCORE 3.2 (for example, EVS with caDSR), you will need to continue to work with the caCORE 3.2 release until the other caCORE 4.0 components are available.

The LexBIG object model was developed by the Mayo Clinic. The associated API, in its native form, assumes a "local" non-distributed means of access. With caCORE EVS 4.0, a proxy layer is provided that enables EVS API clients to access the native LexBIG API from anywhere, without needing to worry about the underlying data sources. This is called the Distributed LexBIG (DLB) API.

The DLB Adapter is another option for caCORE EVS 4.0 clients who choose to interface directly with the LexBIG API. It is essentially a set of convenience methods intended to simplify the use of the LexBIG API (for example, a series of method calls against the DLB API might equate to a single method call to the DLB Adapter).

Note: The DLB Adapter is not intended to represent a complete set of convenience methods. As part of the caCORE EVS 4.0 release, the intention is that users will work with the DLB API and identify/suggest useful methods of convenience to the EVS Development Team.

EVS 3.2 Object Model

The EVS 3.2 Java API is based on the EVS 3.2 object model. The UML Class diagram (object model) in Figure 5-2 provides an overview of the EVS 3.2 domain object classes. The DescLogicConcept and MetaThesaurusConcept are two central Concept classes in the model, with most of the other classes organizing themselves around these entities. The Vocabulary and SecurityToken were added as part of the caCORE 3.2 release. The SecurityToken class can be used to specify security credentials like username, password, security token etc.

The DAO Security model provides data level security to Vocabularies. The MedDRASecurity class, the class that implements the DAOSecurity interface, validates a token against the MedDRA vocabulary and prevents unauthorized users from performing any of the queries against MedDRA. To access MedDRA via the EVS 3.2 Java API, a user must obtain a valid token from NCI CBIIT.

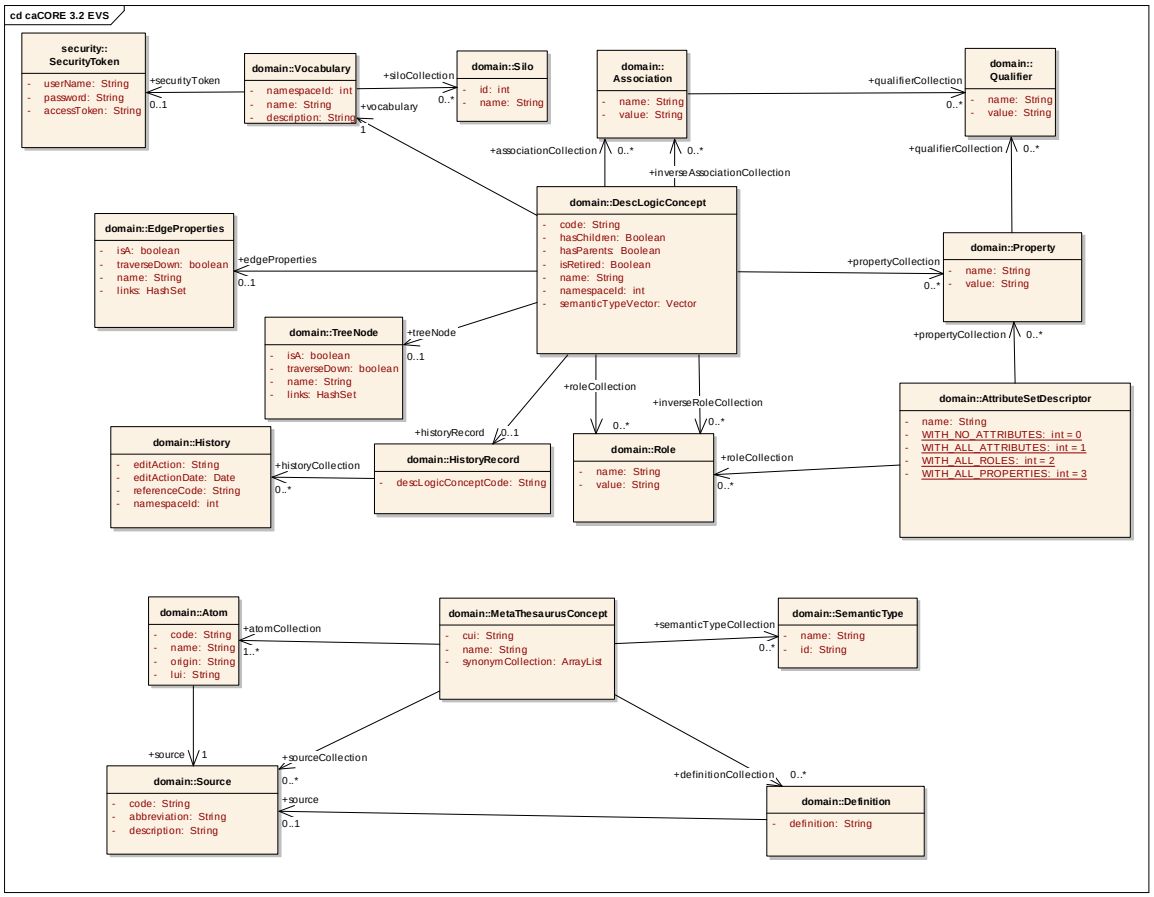

Figure 5-2 The caCORE EVS 3.2 Java API domain object classes

Note: Since the EVS API is generated using the SDK, it is useful to note that the EVS API diverges somewhat from the other caCORE domain models(that is, caDSR and caBIO) in its

search mechanisms. While the other APIs have direct access to their databases, the EVS API does not. Since all EVS queries are passed through the LexBIG APIs, the search and retrieval capabilities are effectively proscribed by the features implemented by the open terminology server.

EVS 3.2 Domain Object Catalog

The caCORE EVS domain objects are implemented as Java beans in the `gov.nih.nci.evs.domain` package. Table 5-1 lists each class and a description. Detailed descriptions about each class and its methods are present in the caCORE EVS 4.0 JavaDocs. The only interface implemented by the EVS domain objects is `java.io.serializable`.

EVS Domain Object	*Description*
Association	Relates a concept or a term to another concept or term. Association falls into three categories; concept association, term association, and synonyms, which are concept-term associations.
Atom	An occurrence of a term in a source.
AttributeSetDescriptor	set of concept attributes that should be retrieved by a given operation.
Definition	Textual definition from an identified source
DescLogicConcept	the fundamental vocabulary entity in the NCI Thesaurus.
EdgeProperties	Specifies the relationship between a concept and its immediate parent when a DefaultMutableTree is generated using the getTree method.
EditActionDate	Stores edit action and date information. This class is deprecated and will be removed from a future release. Please use History class instead.
History	Stores the concept history information.
HistoryRecord	Stores the DescriptionLogicConcept code.
MetaThesaurusConcept	fundamental vocabulary entity in the NCI MetaThesaurus
Property	an attribute of a concept. Examples of properties are "Synonym", "Preferred_Name", "Semantic_Type" etc.
Qualifier	Attached to associations and properties of a concept.
Role	Defines a relationship between two concepts.
SemanticType	a category defined in the semantic network that can be used to group similar concepts
Silo	A repository of customized concept terminology data from a knowledge base. There can be a single silo or multiple silos, each consisting of semantically related concepts and extracted character strings associated with those concepts.
SecurityToken	Stores security information for a Vocabulary.

EVS Domain Object	*Description*
Source	The source is a knowledge base.
TreeNode	Specifies the relationship between a concept and its immediate parent when a DefaultMutableTree is generated using the getTree method. This class is deprecated and will be removed from a future release. Please use EdgeProperties instead.
Vocabulary	Vocabulary entity or namespace.

Table 5-1 caCORE EVS domain objects and descriptions

EVS Data Sources

The EVS data source is the open source, LexBIG terminology server. EVS clients interface with the LexBIG API to retrieve desired vocabulary data.

The EVS provides NCI with services and resources for controlled biomedical vocabularies, and includes both the NCI Thesaurus and the NCI Metathesaurus. The NCI Thesaurus is composed of over 27,000 concepts represented by about 78,000 terms. The Thesaurus is organized into 18 hierarchical trees covering areas such as Neoplasms, Drugs, Anatomy, Genes, Proteins, and Techniques. These terms are deployed by NCI in its automated systems for uses such as key wording and database coding. The NCI Metathesaurus maps terms from one standard vocabulary to another, facilitating collaboration, data sharing, and data pooling for clinical trials and scientific databases. The Metathesaurus is based on the NLM's Unified Medical Language System (UMLS) and is composed of over 70 biomedical vocabularies.

EVS 3.2 Java API

The EVS 3.2 Java API bundled with the caCORE EVS 4.0 release provides direct access to domain objects and all service methods. Because caCORE EVS is natively built in Java, this API provides the fullest set of features and capabilities.

Note: The caCORE 3.2 release also provides an EVS 3.2 Java client API. The difference between the 3.2 and the 4.0 clients is the backend terminology server. caCORE 3.2 uses the proprietary Apelon DTS and caCORE EVS 4.0 uses LexBIG. The API is the same and should only require minor updates to a client application wanting to migrate to the EVS 3.2 Java API provided with caCORE EVS 4.0.

Installation and Configuration

The caCORE EVS Java 3.2 API uses the following software on the client machine (Table 5-2).

Software	*Version*	*Required?*
Java 2 Platform Standard Edition Software 5.0 Development Kit (JDK 5.0)	1.5.0 or higher	Yes
Apache Ant	1.6.5 or higher	Yes

Table 5-2 caCORE EVS Java API Client software

Accessing the caCORE EVS system also requires an Internet connection.

To use the Java API, download the client package provided on the NCICB web site (Figure 5-3).

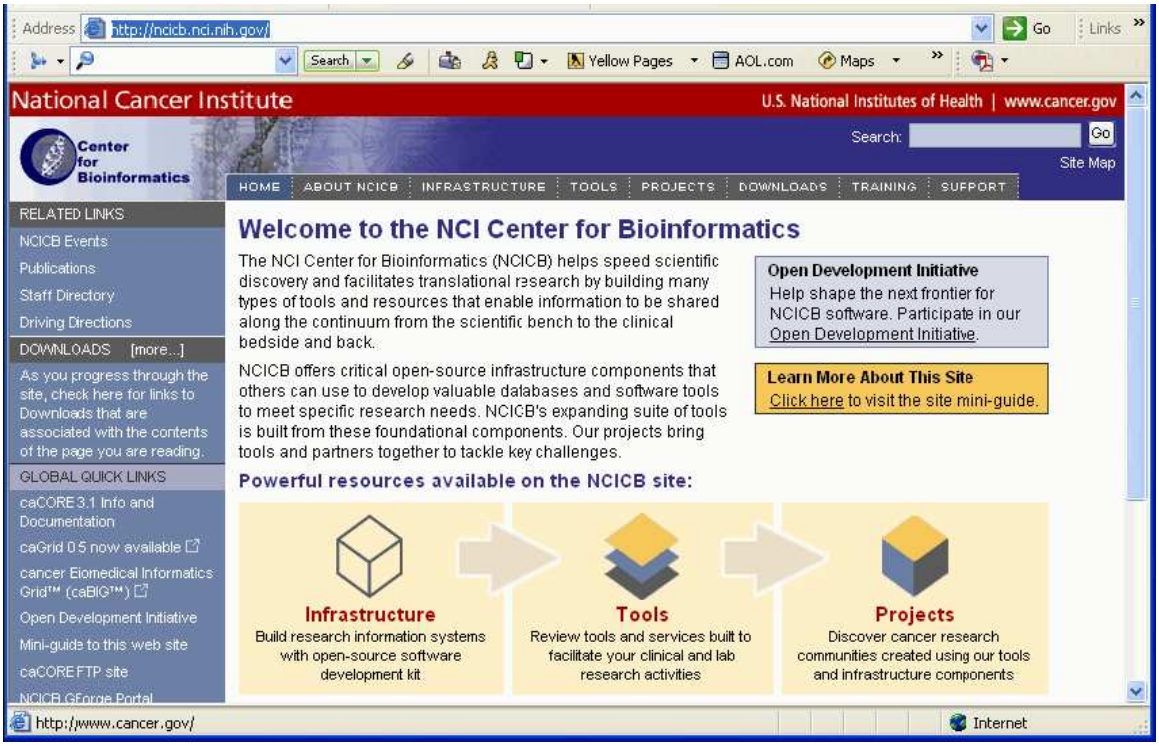

Figure 5-3 Downloads section on the NCICB website

1. Open your browser and go to http://ncicb.nci.nih.gov.
2. Click the Download link on the menu bar.
3. Scroll down to the section titled EVS and click on the Download link.
4. In the provided form, enter your name, email address and institution name and click to Enter the Download Area.
5. Accept the license agreement.
6. On the caCORE EVS downloads page, download the EVS Zip file from the Primary Distribution section.
7. Extract the contents of the downloadable archive to a directory on your hard drive (for example, `c:\evsapi` on Windows or `/usr/local/evsapi` on Linux). The extracted directories and files include the following (Table 5-3):

Directories and Files	*Description*	*Component*
build.xml	Ant build file	Build file
TestClient.java	Java API client samples (for local, remote and web service clients)	Sample code
TestEVS.java	Java API EVS client sample	

Directories and Files	***Description***	***Component***
TestXMLClient.java	XML utility sample	
lib directory	contains required jar files	
evsapi-client.jar	domain objects	
spring.jar	Spring framework	HTTP Remoting
acegi-security-1.0.4	Spring Security	
asm.jar		
antlr-2.7.6	Apache Ant	
axis.jar	Apache Axis	Web services (Java implementation)
saaj.jar	SOAP API for Java	
jaxrpc.jar	Java API for XML-based RPC	
wsdl4j-1.5.1.jar	WSDL for Java	
log4j-1.2.13.jar	logging utilities	Logging
commons-logging-1.1.jar		
commons-codec-1.3.jar		
commons-collections-3.2.jar		
commons-discovery-0.2.jar		
commons-pool-1.3.jar		
evsapi-beans.jar	EVS API Beans	Domain Classes
evsapi-framework.jar	caCORE EVS Framework	
lg*	LexGRID Classes	LexBIG
lb*	LexBIG Classes	
lucene-core-2.0.0	Index Search	LexBIG
castor-1.0.2.jar	Castor serializer/deserializer	XML conversion
xercesImpl.jar	Apache Xerces XML parser	
**.xsd*	XML schemas for objects	
activation.jar		
cglib-2.1.3.jar		
xml.properties		
xml-mapping.xml		
conf directory		
remoteService.xml		

Directories and Files	Description	Component
deploy.wsdd		
log4j.properties	Logging utilities configuration properties	

Table 5-3 Extracted directories and files in caCORE EVS client package

All of the jar files in the lib directory of the caCORE EVS client package, in addition to the files in the conf directory, are required to use the Java API. These should be included in the Java classpath when building applications. The `build.xml` file that is included demonstrates how to do this when using Ant for command-line builds. If you are using an integrated development environment (IDE) such as Eclipse, refer to the tool's documentation for information on how to set the classpath.

Installation Verification: A Simple Example

To run the simple example program after installing the caCORE EVS client, open a command prompt or terminal window from the directory where you extracted the down-loaded archive and enter `ant`. This displays a list of ant targets that identify the test execution options as shown below.

```
Buildfile: build.xml


help:
     [echo] ==============================
     [echo] caCORE EVS API - HELP
     [echo] ==============================
     [echo]
     [echo] To run the test programs use the following commands:
     [echo]
     [echo] 1. TestClient -  ant run
     [echo] 2. TestEVS    -  ant runevs
     [echo] 3. TestXML    -  ant runxml
     [echo]


BUILD SUCCESSFUL
Total time: 0 seconds
```

Enter the desired ant command and the associated test class is compiled and run. Successfully running this example code indicates that you have properly installed and configured the caCORE EVS client. The following is a short segment of code from the TestClient class along with an explanation of its functioning.

```
EVSApplicationService appService =
   (EVSApplicationService)ApplicationServiceProvider.
```

```
      getApplicationService();
try {
   DescLogicConcept dlc = new DescLogicConcept();
   dlc.setName("ear*");
   Collection results = appService.search("gov.nih.nci.evs.domain.DescLogicConcept",
   dlc);
   System.out.println("Results: "+ results.size());
   for(Object o : results) {
         DescLogicConcept obj = (DescLogicConcept)o;
         System.out.println("Concept name : "+ obj.getName() +"\t"+ obj.getCode());
         Vector propList = (Vector)obj.getPropertyCollection();
         if (propList == null) {
                        System.out.println("NO properties found");
         } else {
                        System.out.println("Properties: "+ propList.size());
         }

                Vector roleList = (Vector) obj.getRoleCollection();
                if (roleList == null) {
                        System.out.println("No roles found");
                } else {
                        System.out.println("Roles: "+ roleList.size());
                }

   }
} catch(Exception e) {
  System.out.println(">>>"+e.getMessage());
  e.printStackTrace();
}
```

This code snippet creates an instance of a class that implements the EVSApplicationService interface. This interface defines the service methods used to access data objects. A criterion object is then created that defines the attribute values for which to search. The search method of the EVSApplicationService implementation is called with parameters that indicate the type of objects to return, `gov.nih.nci.evs.domain.DescLogicConcept`, and the criteria that returned objects must meet, defined by the `dlc` object. The search method returns

objects in a Collection, which is iterated through to print some basic information about the objects.

Although this is a fairly simple example of the use of the EVS Java API, a similar sequence can be followed with more complex criteria to perform sophisticated manipulation of the data provided by caCORE EVS. Additional information and examples are provided in the sections that follow.

Search Paradigm

The caCORE EVS architecture includes a service layer that provides a single, common access paradigm to clients using any of the provided interfaces. As an object-oriented middleware layer designed for flexible data access, caCORE EVS relies heavily on strongly typed objects and an object-in/object-out mechanism. The basic order of operations required to access and use a caCORE EVS system is as follows:

1. Ensure that the client application has knowledge of the objects in the domain space.
2. Formulate the query criteria using the domain objects.
3. Establish a connection to the server.
4. Submit the query objects and specify the desired class of objects to be returned.
5. Use and manipulate the result set as desired.

There are four primary application programming interfaces (APIs) native to caCORE EVS systems. Each of the available interfaces described below uses this same paradigm to provide access to the caCORE EVS domain model, but with minor changes relating primarily to the syntax and structure of the clients. The following sections describe each API, identify installation and configuration requirements, and provide code examples.

The sequence diagram in Figure 5-4 provides an overview of the caCORE 4.0 EVS API search mechanism implemented to access the NCI EVS vocabularies.

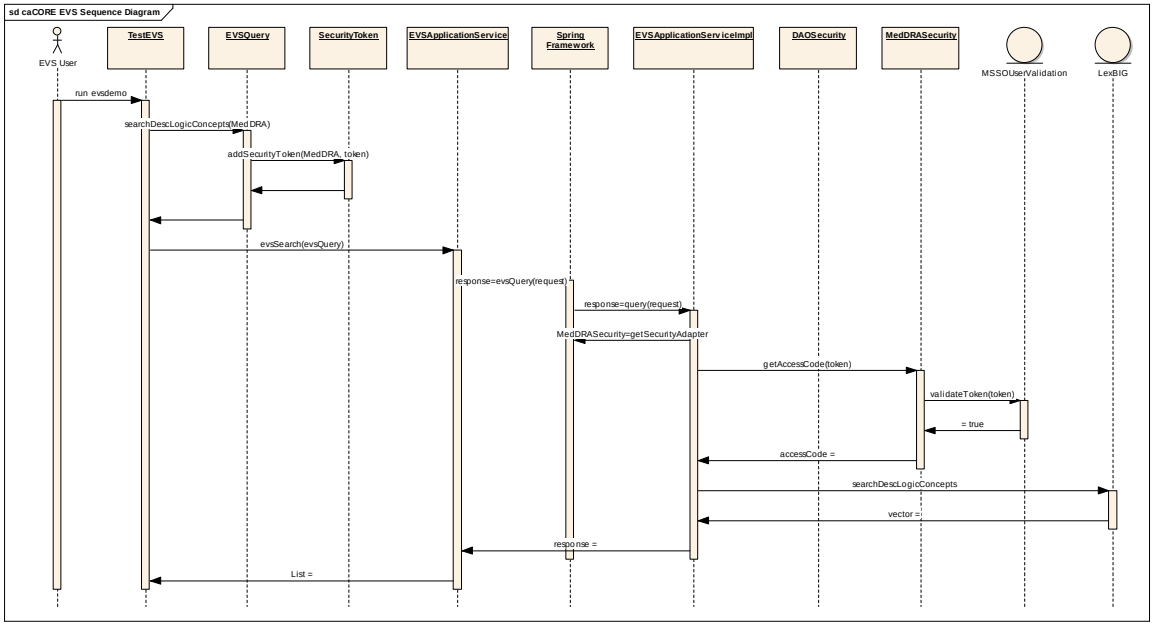

Figure 5-4 EVS sequence diagram

An EVS search is performed by calling the *evsSearch* operation defined in the *ApplicationService* class.

```
List evsSearch(EVSQuery evsQuery);
```

EVSQuery and EVSQueryImpl

The `gov.nih.nci.evs.query` package consists of the `EVSQuery.java` interface and the `EVSQueryImpl.java` class. The methods defined in the `EVSQuery.java` file can be used to query the LexBIG Terminology Server. The query object generated by this class can hold one query at a time. The following example code segment demonstrates an `EVSQuery` object that calls the `searchDescLogicConcept` method.

```
String vocabularyName = "GO";

String conceptCode = "GO:0005667";

EVSQuery evsQuery = new EVSQueryImpl();

evsQuery.searchDescLogicConcept(vocabularyName,conceptCode,true);
```

To perform a search on the Description Logic Vocabulary you must specify the vocabulary name. In most instances methods that do not require vocabulary names are NCI MetaThesaurus queries.

EVSQuery Methods and Parameters

Most of the methods defined in the `EVSQuery` accept concept names or concept codes. If a vocabulary name is required as a parameter along with a concept code or name, a valid `DescLogicConcept` name or code needs to be passed to the search method.

Note: A search term is a String and is not considered as a valid concept name. To get a valid *DescLogicConcept* name you must perform a search using the *searchDescLogicConcept* method. Likewise to get a valid *MetaThesaurusConcept* name or *CUI* (Concept Unique Identifier) you must perform a search using the *searchMetaThesaurus* method. Most of the search methods defined in the *EVSQuery* require a valid concept name or code.

Some of the methods defined in the `EVSQuery` are listed in the following table (Table 5-4).

Method name	***Parameter***	***Comments***	***Returned by evsSearch***
searchDescLogicConcept	String vocabularyName	A valid Description Logic vocabulary name such as "NCI_Thesaurus", "GO", "HL7" etc	Returns one or more DescLogicConcepts in a List.
	String searchTerm	Any string value	
	int limit	Maximum number of records	
searchMetaThesaurus	String searchTerm	Any String value or a valid Concept Unique Identifier. A Concept Unique identifier is used to uniquely	Returns one or more MetaThesaurusConcepts in a List.

Method name	Parameter	Comments	Returned by evsSearch
		identify concepts in the MetaThesaurus.	
	int Limit	Maximum number of records	
	String Source	Source abbreviation. Each Source has a source abbreviation that can uniquely identify a source.	
	boolean Cui	This value is set to true if a concept unique identifier is used as a search term	
	boolean shortResponse	Set to true for short response	
	boolean score	Set to true for score	
getHistoryRecords	String vocabularyName	A valid Description Logic vocabulary name such as "NCI_Thesaurus", "GO", "HL7" etc	Returns one or more HistoryRecords in a List.
	String conceptCode	A valid code of a DescLogicConcept.	
getVocabularyNames			Returns one or more Description Logic vocabulary names in a List
getMetaSources			Returns one or more Source objects in a List.
searchSourceByCode	String code	A valid Atom code.	Returns one or more MetaThesaurusConcepts in a List.
	String sourceAbbreviation	Valid source abbreviation	
getTree	String vocabularyName	A valid Description Logic vocabulary name	Returns a DescLogicConcept tree in a List
	String rootName	A valid DescLogicConcept name	
	boolean direction	Set to true if traverse down	
	boolean isaFlag	Set to true if relationship is child	
	int attributes	Sets a AttributeSetDescriptor value	
	int levels	Depth of the tree	
	Vector roles	Valid role names	

Table 5-4 Methods defined in the EVSQuery

Accessing Secured Vocabularies

MedDRA is a Secured vocabulary for which a user must obtain a valid security token to access. The example below depicts the syntax of setting a security token to access a secured Vocabulary.

```
gov.nih.nci.evs.query.EVSQuery evsQuery = new
gov.nih.nci.evs.query.EVSQueryImpl();

gov.nih.nci.evs.security.SecurityToken token = new
gov.nih.nci.evs.security.SecurityToken();

token.setAccessToken(“123456”);

evsQuery.addSecurityToken(“MedDRA”, token);

evsQuery.getDescLogicConcept(“MedDRA”, “Blood”, false);
```

Note: You must obtain a valid security token from NCICB to access MedDRA via the caCORE EVS API. The security token value used in the example is not valid.

Use the following instructions to create an EVS search request.

1. Create an *ApplicationService* instance.

   ```
   EVSApplicationService appService = (EVSApplicationService)
   ApplicationServiceProvider.getApplicationService();
   ```

2. Instantiate an *EVSQuery* instance and set the method name and parameters.

   ```
   EVSQuery evsQuery = new EVSQueryImpl();

   evsQuery.searchDescLogicConcepts("NCI_Thesaurus","blood*",10);
   ```

3. Set the security token value. This step can be omitted if the vocabulary does not require a security token.

   ```
   gov.nih.nci.evs.security.SecurityToken token = new
   gov.nih.nci.evs.security.SecurityToken();

   token.setAccessToken("xxxxxx");

   evsQuery.addSecurityToken(vocabularyName, token);
   ```

4. Call the *evsSearch* method defined in the *ApplicationService* class to query EVS.

   ```
   List evsResults = (List)appService.evsSearch(evsQuery);
   ```

5. The result objects are populated. The return type varies based on the search method call set in the `EVSQuery` instance.

Examples of Use

Example One: Search for DescLogicConcepts by Term

```
1  public static void main(String[] args) {
```

```
2     try {
3         EVSApplicationService appService =
4             (EVSApplicationService)ApplicationServiceProvider.
5                   getApplicationService();
6         EVSQuery evsQuery = new EVSQueryImpl();
7         evsQuery.searchDescLogicConcepts("NCI_Thesaurus","blood*",10);
8         List evsResults = (List)appService.evsSearch(evsQuery);
9
10    } catch(ApplicationException ex){
11    }
12
13 }
```

Lines	Description
3	Creates an instance of a class that implements the ApplicationService interface; this interface defines the service methods used to access data objects.
4	Creates a new EVSQuery object.
5	Specifies the search method and parameters. The searchDescLogicConcept method performs a search in the "NCI_Thesaurus" vocabulary for a term that starts with "blood" and returns a maximum of ten Concepts if found.
6	Calls the evsSearch method of the ApplicationService implementation passing the EVSQuery object. This method returns a List Collection. The type of object that is returned depends on the search parameters set in the EVSQuery object; in this case the searchDescLogicConcept method was invoked, the resulting objects are of type DescLogicConcept.

Example Two: Search MetaThesaurusConcepts by Atom

```
1 try {
2     EVSApplicationService appService =
        (EVSApplicationService)ApplicationServiceProvider.
           getApplicationService();
3     EVSQuery evsQuery = new EVSQueryImpl();
4     evsQuery.searchSourceByAtomCode("10834-0","*");
5     List evsResults = (List)appService.evsSearch(evsQuery);
6     for(int m=0; m<evsResults.size(); m++ ){
7        MetaThesaurusConcept concept =
         (MetaThesaurusConcept)evsResults.get(m);
```

```
8        System.out.println("\nConcept code: "+concept.getCui()
         +"\n\t"+concept.getName());
9        List sList = concept.getSourceCollection();
10       System.out.println("\tSource-->" + sList.size());
11       for(int y=0; y<sList.size(); y++){
12           Source s = (Source)sList.get(y);
13           System.out.println("\t - "+s.getAbbreviation());
21       }
14       List semanticList = concept.getSemanticTypeCollection();
15       System.out.println("\tSemanticType---> count ="+
         semanticList.size());
16       for(int z=0; z<semanticList.size(); z++){
17          SemanticType sType = (SemanticType) semanticList.get(z);
18          System.out.println("\t- Id: "+sType.getId()+"\n\t- Name:
            "+sType.getName());
19       }
20       List atomList = concept.getAtomCollection();
21       System.out.println("\tAtoms -----> count = "+ atomList.size());
22       for(int i=0;i<atomList.size(); i++){
23           Atom at = (Atom)atomList.get(i);
24           System.out.println("\t -Code: "+ at.getCode()+" -Name: "+
             at.getName()
25                +" -LUI: "+ at.getLui()+" -Source: "+
                  at.getSource().getAbbreviation());
26       }
27       List synList = concept.getSynonymCollection();
28       System.out.println("\tSynonyms -----> count = "+
         synList.size());
29       for(int i=0; i< synList.size(); i++){
30           System.out.println("\t - "+ (String) synList.get(i));
31       }
32    }
33 } catch(ApplicationException ex){
34
```

```
35 }
```

Lines	Description
2	Creates an instance the EVSApplicationService.
3	Creates a new EVSQuery object.
4	Specifies the search method and parameters. The searchSourceByAtomCode method performs a search on all the sources specified in the MetaThesaurus for MetaThesaurusConcepts that has an Atom code value "10834-0 ". The source abbreviation specified is "*"; therefore all sources are searched for the Atom specified.
5	Calls the evsSearch method of the EVSApplicationService implementation passing the EVSQuery object. This method returns a List Collection. The type of object that is returned depends on the search parameters set in the EVSQuery object; in this case the searchSourceByAtomCode method was invoked, the resulting objects are of type MetaThesaurusConcept.
6	Traverse through the result set.
7	Cast the result object to a MetaThesaurusConcept.
6-31	Prints the attributes and association values of the MetaThesaurusConcept.

Web Services API

The caCORE EVS Web services API allows access to caCORE EVS data from development environments where the Java API cannot be used, or where use of XML Web services is more desirable. This includes non-Java platforms and languages such as Perl, C/ C++,.NET Framework (C#, VB.Net), Python, etc.

The caCORE EVS Web services API allows access to vocabulary data from development environments where the Java API cannot be used, or where use of XML Web services is more desirable. This includes non-Java platforms and languages such as Perl, C/ C++,.NET Framework (C#, VB.Net), Python, etc.

The Web services interface can be used in any language-specific application that provides a mechanism for consuming XML Web services based on the Simple Object Access Protocol (SOAP). In these environments, connecting to caCORE EVS can be as simple as providing the endpoint URL. Some platforms and languages require additional client-side code to handle the implementation of the SOAP envelope and the resolution of SOAP types. A list of packages catering to different programming languages is available at http://www.w3.org/TR/SOAP/ and at http://www.soapware.org/. To maximize standards-based interoperability, the caCORE Web service conforms to the Web Services Interoperability Organization (WS-I) Basic Profile. The WS-I Basic Profile provides a set of non-proprietary specifications and implementation guidelines enabling interoperability between diverse systems. More information about WS-I compliance is available at http://www.ws-i.org. On the server side, Apache Axis is used to provide SOAP-based inter-application communication. Axis provides the appropriate serialization and deserialization methods for the Java beans to achieve an application-independent interface. For more information about Axis, visit http://ws.apache.org/axis/.

Configuration

The caCORE/EVS WSDL file is located at http://evsapi.nci.nih.gov/evsapi40/services/evsapi40Service?wsdl . In addition to describing the protocols, ports and operations exposed by the caCORE EVS Web service, this file can be used by a number of IDEs and tools to generate stubs for caCORE EVS objects. This allows code on different platforms to instantiate objects native to each for use as parameters and return values for the Web service methods. Consult the specific documentation for your platform for more information on how to use the WSDL file to generate class stubs. The caCORE EVS Web services interface has a single endpoint called evsapiService, which is located at http://evsapi.nci.nih.gov/evsapi40/services/evsapi40Service. Client applications should use this URL to invoke Web service methods.

Operations

Through the caCOREService endpoint, developers have access to three operations:

Operation	`getVersion`
Input Schema	`None`

Output Schema	`<complexType>`
	`<sequence>`
	`<element type="xsd:string"/>`
	`</sequence>`
	`</complexType>`

Description	Returns an xsd:string containing the version of the running caCORE system (e.g., "caCORE 4.0")

Operation	`queryObject`

Input Schema	`<complexType>`
	`<sequence>`
	`<element name="in0" type="xsd:string"/> <element name="in1" type="xsd:anyType"/>`
	`</sequence>`
	`</complexType>`

Output Schema	`<sequence>`
	`<element name="queryReturn"type= "ArrayOf_xsd_anyType"/>`
	`</sequence>`

Description	Performs a search for objects conforming to the criteria defined by input parameter in1 and whose resulting objects are of the type reached by traversing the node graph specified by parameter

	in0; the result is a set of serialized objects (the type ArrayOf_xsd_anyType resolves to a sequence of xsd:anyType elements)

Operation	Query
Input Schema	<complexType> <sequence> <element name="in0" type="xsd:string"/> <element name="in1" type="xsd:anyType"/> <element name="in2" type="xsd:int"/> <element name="in3" type="xsd:int"/> </sequence> </complexType>
Output Schema	<sequence> <element name="queryReturn" type="ArrayOf_xsd_anyType"/> </sequence>
Description	Identical to the previous queryObject method, but allows for control over the result set by specifying the row number of the first row (in2) and the maximum number of objects to return (in3)

Developers should be aware of a significant behavioral decision that has been made regarding the Web services interface. When a query is performed with this interface, returned objects do not contain or refer to their associated objects (a notable exception is with the EVS domain model-see below). This means that a separate query invocation must be performed for each set of associated objects that need to be retrieved. One of the examples below demonstrates this functionality.

Considerations

The EVS domain objects are unique in the way they are used with the Web services interface. EVS classes that can be queried from Web services always provide associations to their related objects. This enables access to the objects that are not of type *DescLogicConcept*, *MetaThesaurusConcept,* or *HistoryRecord*.

Because of the unique behavior and properties of the EVS domain model, queries using the Web services interface can be performed only on the selected attribute values listed in Table 5-5.

Class	***Available search attributes***
DescLogicConcept	Name
	code

Class	*Available search attributes*
	Property name and value
	Role name and value
MetaThesaurus Concept	Name
	cui (concept unique identifier)
	Atom code and Source abbreviation
HistoryRecord	DescLogicConcept name or code (HistoryRecord is the targetObject and the DescLogicConcept is the criteriaObject)

Table 5-5 Allowable attributes for searching the EVS domain model

Examples of Use

Example One: Simple Search (NCI Thesaurus)

The following code demonstrates a simple query written in the Java language that uses the Web services API. This example uses Apache Axis on the client side to handle the type mapping, SOAP encoding, and operation invocation.

```
1   try {
2       String endpointURL
        "http://evsapi.nci.nih.gov/evsapi40/services/evsapi40Service";
3       String methodName = "queryObject";
4       Service service = new Service();
5       Call call = (Call) service.createCall();
6
7       call.setTargetEndpointAddress(new java.net.URL(endpointURL));
8       call.setOperationName(new QName("EVSService", "queryObject"));
9       call.addParameter("arg1",
        org.apache.axis.encoding.XMLType.XSD_STRING, ParameterMode.IN);
10      call.addParameter("arg2",
        org.apache.axis.encoding.XMLType.XSD_ANYTYPE, ParameterMode.IN);
11      call.setReturnType(org.apache.axis.encoding.XMLType.SOAP_ARRAY);
12
13      QName qnDLCArr = new QName("urn:domain.evs.nci.nih.gov",
        "ArrayOf_tns1_DescLogicConcept");
14      call.registerTypeMapping(DescLogicConcept.class, qnDLCArr,
15          new org.apache.axis.encoding.ser.BeanSerializerFactory(),
16          new org.apache.axis.encoding.ser.BeanDeserializerFactory());
17
18      DescLogicConcept dlc = new DescLogicConcept();
19      dlc.setName("ear*");
20
21      call.setReturnType(qnDLCArr);
22
23      Object thesarusParams = new
        Object[]{"gov.nih.nci.evs.domain.DescLogicConcept", dlc};
24      DescLogicConcept[] dlcs =
        (DescLogicConcept[])call.invoke(thesaurusParams);
```

```
25
26      char counter = 'a';
27
28      for (DescLogicConcept d: dlcs) {
29          System.out.println("\t" + counter + ") Concept name; " +
            d.getName());
30          System.out.println("\t code: " + d.getCode());
31          System.out.println("\t ---------------------------------");
32          counter++;
33      }
34      System.out.println("\tNumber of items returns from Thesaurus " +
        dlcs.length);
35   } catch (Exception ex) {
36       System.out.println(ex.getMessage());
37   }
```

Lines	*Description*
4 – 5	Defines a new Web Service Call
6-9, 20	Sets the call properties including the name of the operation to invoke, the target property address, the input parameters that will be sent and the return type
12 – 15	Maps a serialized object to it's java equivalent using the qualified name of the object from the WSDL file; in this case, the XML element urn:domain.evs.nci.nih.gov namespace is mapped to the Java DescLogicConcept Array
17 – 18	Creates a DescLogicConcept and sets the name attribute to "ear*".
22- 23	Invokes the Web Service operation using an array of two objects (target class name and criteria object) as input parameters and expecting an object array as its result.
26	Cast each object in the result array to type DescLogicConcept and print

Example Two: EVS Domain Search (NCI MetaThesaurus)

The code below demonstrates use of the Web Services interface to query data from the NCI MetaThesaurus using EVS domain objects.

```
1   try {
2       String endpointURL =
        "http://evsapi.nci.nih.gov/evsapi40/services/evsapi40Service";
3       String methodName = "queryObject";
4       Service service = new Service();
5       Call call = (Call) service.createCall();
6
7       call.setTargetEndpointAddress(new java.net.URL(endpointURL));
8       call.setOperationName(new QName("EVSService", "queryObject"));
9       call.addParameter("arg1",
        org.apache.axis.encoding.XMLType.XSD_STRING, ParameterMode.IN);
10      call.addParameter("arg2",
        org.apache.axis.encoding.XMLType.XSD_ANYTYPE, ParameterMode.IN);
11      call.setReturnType(org.apache.axis.encoding.XMLType.SOAP_ARRAY);
12
```

```
13      QName qnMTCArr = new QName("urn:domain.evs.nci.nih.gov",
        "ArrayOf_tns1_MetaThesaurusConcept");
14      call.registerTypeMapping(MetaThesaurusConcept.class, qnMTCArr,
15          new org.apache.axis.encoding.ser.BeanSerializerFactory(),
16          new org.apache.axis.encoding.ser.BeanDeserializerFactory());
17
18      MetaThesaurusConcept mtc = new MetaThesaurusConcept();
19      MTC.setName("blood*");
20
21      call.setReturnType(qnMTCArr);
22
23      Object metaParams = new
        Object[]{"gov.nih.nci.evs.domain.MetaThesaurusConcept", mtc};
24      MetaThesaurusConcept[] meta = null;
25
25      try {
26          meta = (MetaThesaurusConcept[])call.invoke(metaParams);
26          char counter = 'a';
27
28          for (MetaThesaurusConcept m: meta) {
29              System.out.println("\t" + counter + ") Concept name; " +
                m.getName());
30              System.out.println("\t code: " + m.getCui());
31              System.out.println("\t ---------------------------------
                ");
32              counter++;
33          }
34          System.out.println("\tSize" + meta.length);
35     } catch (Exception ex) {
36          System.out.println("Error: " + ex);
37     }
38    } catch (Exception ex) {
39        System.out.println(ex.getMessage());
40    }
```

Lines	***Description***
4 – 5	Defines a new Web Service Call
6-9, 18	Sets the call properties including the name of the operation to invoke, the target property address, the input parameters that will be sent and the return type
11 – 14	Maps a serialized object to it's java equivalent using the qualified name of the object from the WSDL file; in this case, the XML element urn:domain.evs.nci.nih.gov namespace is mapped to the Java MetaThesaurusConcept Array
16 – 17	Creates a MetaThesaurusConcept and sets the name attribute to "blood*".
23	Invokes the Web Service operation using an array of two objects (target class name and criteria object) as input parameters and expecting an object array as its result.
25	Cast each object in the result array to type MetaThesaurusConcept and print

Limitations

By default, the queryObject operation limits the result set to 1000 objects, even if the size of the result set is larger. To retrieve the objects past the 1000th record, you must use the query operation and specify the first object index (parameter in2) to be greater than 1000.

Result sets serialized and returned by the Web services interface do not currently include associations to related objects. A consequence of this behavior is that nested criteria objects with one-to-many associations that are passed to the query or queryObject operations will result in an exception being thrown.

Because the Web services invocation has an inherent timeout behavior, queries that take a long time to execute may not complete. If this is the case, use the query method to specify a smaller result count.

Access to the EVS domain model is limited by the Web services interface, as shown in the following table (Table 5-6).

Typical Behavior	***EVS Model Behavior***
Can query for any object in the object model	Can query only for a DescLogicConcept, HistoryRecord or a MetaThesaurusConcept
The association values of the caCORE domain objects are not populated; need to run a second query to get associated values	All attributes of the result object are populated
Can perform queries on any attribute value	Queries can be performed only on selected attribute values (see Table 5-4)

Table 5-6 Access to the EVS domain model

XML-HTTP API

The caCORE EVS XML-HTTP API, based on the REST (Representational State Transfer) architectural style, provides a simple interface using the HTTP protocol. In addition to its ability to be invoked from most internet browsers, developers can use this interface to build applications that do not require any programming overhead other than an HTTP client. This is particularly useful for developing web applications using AJAX (asynchronous JavaScript and XML).

Service Location and Syntax

The caCORE EVS XML-HTTP interface uses the following URL syntax (Table 5-7):

```
http://{server}/{servlet}?query={returnClass}&{criteria}&
      resultCounter={counter}&startIndex={index}&
      pageSize={pageSize}&pageNumber={pageNumber}
```

Element	*Meaning*	*Required*	*Example*
server	Name of the web server on which caCORE EVS 4.0 web application is deployed.	Yes	evsapi.nci.nih.gov/evsapi40
servlet	URI and the name of the servlet that will accept the HTTP GET requests	Yes	evsapi40 /GetXML evsapi40 /GetHTML
returnClass	Class name indicating the type of objects that this query should return	Yes	query=DescLogicConcept
criteria	Search request criteria describing the requested objects	Yes	DescLogicConcept [@id=2]
counter	Number of top level objects returned by the search	No	resultCounter=500
index	Start index of the result set	No	startIndex=25
pageSize	Number of records to display on each "page	No	pageSize=50
pageNumber	The number of the "page" for which to display results	No	pageNumber=3

Table 5-7 URL syntax used by the caCORE EVS XML-HTTP interface

The caCORE EVS architecture currently provides two servlets that accept incoming requests:

- **GetXML** returns results in an XML format that can be parsed and consumed by most programming languages and many document authoring and management tools.

- **GetHTML** presents result using a simple HTML interface that can be viewed by most modern Internet browsers.

Within the request string of the URL, the criteria element specifies the search criteria using XQuery-like syntax. Within this syntax, square brackets ("[" and "]") represent attributes and associated roles of a class, the "at" symbol ("@") signals an attribute name/value pair, and a forward slash character ("/") specifies nested criteria. Criteria statements within XML-HTTP queries are generally of the following forms (although more complex statements can also be formed):

```
{ClassName}[@{attributeName}={value}] [@{attributeName}={value}]…
ClassName}[@{attributeName}={value}]/
{ClassName}[@{attributeName}={value}]/…
```

Parameter	*Meaning*	*Example*
ClassName	The name of a class	DescLogicConcept
attributeName	The name of an attribute of the return class or an associated class	name
value	The value of an attribute	ear*

Table 5-8 Criteria statements within XML-HTTP queries

Examples of Use

The following examples demonstrate use of the XML-HTTP interface. In actual use, the queries shown here would either be submitted by a block of code or entered in the address bar of an Internet browser. Also note that the servlet name *GetXML* in each of the examples can be replaced with *GetHTML* to view with layout and markup in a browser.

Query	http://evsapi.nci.nih.gov/evsapi40/GetXML?query=DescLogicConcept[name=blood*]
Semantic Meaning	Find all objects of type DescLogicConcept whose name starts with 'blood'
Biological Meaning	Find all concepts the refer to blood

Working With Result Sets

Because HTTP is a stateless protocol, the caCORE EVS server has no knowledge of the context of any incoming request. Consequently, each invocation of GetXML or GetHTML must contain all of the information necessary to retrieve the request, regardless of previous requests. Developers should consider this when working with the XML-HTTP interface.

Retrieving Related Results using XLinks

When using the GetXML servlet to retrieve results as XML, associations between objects are converted to XLinks within the XML. The link notation, shown below, allows the client to make a subsequent request to retrieve the associated objects.

```
<class name="gov.nih.nci.evs.domain.DescLogicConcept" recordNumber="1">
…
<field name="Vocabulary"
       xlink:type="simple"
       xlink:href="http://evsapi.nci.nih.gov/evsapi40/GetXML?
              Query=Vocabulary&DescLogicConcept[@id=5]"> getVocabulary
</field>
…
</class>
```

Controlling the Number of Items Returned

The GetXML servlet provides a throttling mechanism to allow developers to define the number of results returned on any single request and where in the result set to start. For example, if a search request yields 500 results, specifying resultCounter=450 will return only the last 50 records. Similarly, specifying startIndex=50 will return only the first 50 records.

Paging Results

In addition to controlling the number of results to display, the GetXML servlet also provides a mechanism to support "paging". This concept, common to many web sites, allows results to be displayed over a number of pages, so that, for example, a request that yields 500 objects could be displayed over 10 pages of 50 objects each. When the paging feature is used, the GetXML servlet will include XLinks to each of the result pages in an XML <page/> element. The element data of the <page/> element is the number of the page, suitable for output as text or HTML when using an XSL stylesheet:

```
<page number="1"
      xlink:type="simple"
      xlink:href="http://evsapi.nci.nih.gov/evsapi40/GetXML?query={query}&
pageNumber=4&resultCounter=1000&startIndex=0"> 4
</page>
```

Limitations

When specifying attribute values in the query string, use of the following characters generates an error: [ ] / \ # & %.

Utility Methods

XML Utility

caCORE (as accessible through the SDK framework) provides a utility (*XMLUtility* class) in the gov.nih.nci.common.util package that provides the capability of converting caCORE EVS domain objects between native Java objects and XML serializations based on standard XML schemas. The XML schemas for all domain objects in caCORE EVS, directly generated from the UML model (described earlier), are included in the downloadable archive (in the lib directory). Currently, the XML generated using the *XMLUtility* class includes only the object attributes; associated objects are not included.

Properties used by the XML utility are included in two files. The first, xml.properties, defines some basic information needed by the class and also contains a property defining the filename of the second. This second file, called xml-mapping.xml by default, defines the binding between class and attribute names and the corresponding XML element and attribute names.

A default marshaller and unmarshaller are provided with the caCORE EVS client; developers wishing to use their own should provide the fully-qualified name of the two classes in the xml.properties file.

In the following code, the XML utility is used to serialize an object and save it to a file stream. A new object is then instantiated from the file using the utility.

```
1 ApplicationService appService =
  ApplicationServiceProvider.getApplicationService();
2
3 Marshaller marshaller = new caCOREMarshaller("xml-mapping.xml", false);
4 Unmarshaller unmarshaller = new caCOREUnmarshaller("unmarshaller-xml-
  mapping.xml", false);
5 XMLUtility myUtil = new XMLUtility(marshaller, unmarshaller);
6 Class klass = Vocabulary.class;
7 Object o = klass.newInstance();
8 System.out.println("Searching for "+klass.getName());
9 try {
10     Collection results = appService.search(klass, o);
11     for(Object obj : results) {
12       File myFile = new File("./output/" + klass.getName() +
         "_test.xml");
```

```
13
14       FileWriter myWriter = new FileWriter(myFile);
15       myUtil.toXML(obj, myWriter);
16       myWriter.close();
17       printObject(obj, klass);
18       DocumentBuilder parser = DocumentBuilderFactory
19       .newInstance().newDocumentBuilder();
20       Document document = parser.parse(myFile);
21       SchemaFactory factory = SchemaFactory
22       .newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
23
24       try {
25          System.out.println("Validating " + klass.getName() +
26          " against the schema......\n\n");
27          Source schemaFile = new StreamSource(
28          Thread.currentThread().getContextClassLoader().
            getResourceAsStream(
29          klass.getPackage().getName() + ".xsd"));
30          Schema schema = factory.newSchema(schemaFile);
31          Validator validator = schema.newValidator();
32
33          validator.validate(new DOMSource(document));
34          System.out.println(klass.getName() + " has been
            validated!!!\n\n");
35       } catch (Exception e) {
36          System.out.println(klass.getName() +
37             " has failed validation!!!  Error reason is: \n\n" +
               e.getMessage());
38       }
39
40       System.out.println("Un-marshalling " + klass.getName() + " from "
         +
41       myFile.getName() +"......\n\n");
42       Object myObj = (Object) myUtil.fromXML(myFile);
```

```
43
44        printObject(myObj, klass);
45        myWriter.close();
46        break;
47     }
48 } catch(Exception e) {
49    System.out.println("Exception caught: " + e.getMessage());
50    e.printStackTrace();
51 }
```

Lines	Description
1	Instantiate the EVSApplicationService
3-5	Instantiate the marshaller and unmarshaller using the appropriate mapping files and use to instantiate the XMLUtility
10	Perform a Search against the Vocabulary class
11	Iterate through all of the returned results
12	Instantiate a new xml file based on the search class
14-17	Use the XML utility to convert the returned object to XML and write to the output stream
18-38	Validate the generated XML file
40-51	Unmarshall the generated XML object from the writen file and print to System.out

Distributed LexBIG API

Overview

caCORE EVS is making a gradual transition towards a pure LexBIG backend terminology server and exposure of the LexBIG Service object model in place of the existing EVS 3.2 object model. caCORE 3.2 and earlier required a custom API to sit between external users of the system and the proprietary Apelon Terminology Server APIs. With the transition to LexBIG, caCORE EVS can publicly expose the open source terminology service API without the requirement of a custom API layer.

To allow users of caCORE EVS 3.2 and earlier to begin to prepare for transition to the LexBIG API, caCORE EVS 4.0 provides a Distributed LexBIG (DLB) API in addition to the EVS API (based on the 3.2 object model). The DLB API provides remote access to the LexBIG API residing on the caCORE EVS server.

Architecture

The LexBIG API is exposed by the EVS caCORE System for remote, distributed access (Figure 5-5). The caCORE System's 'EVSApplicationService' class implements the

'LexBIGService' interface, effectively exposing LexBIG via caCORE. Since the objects returned from the LexBIGService are not merely beans, but full fledged Data Access Objects (DAOs) in many cases, the caCORE EVS client is configured to proxy method calls into the LexBIG objects and forward them to the caCORE server so that they execute within the LexBIG environment.

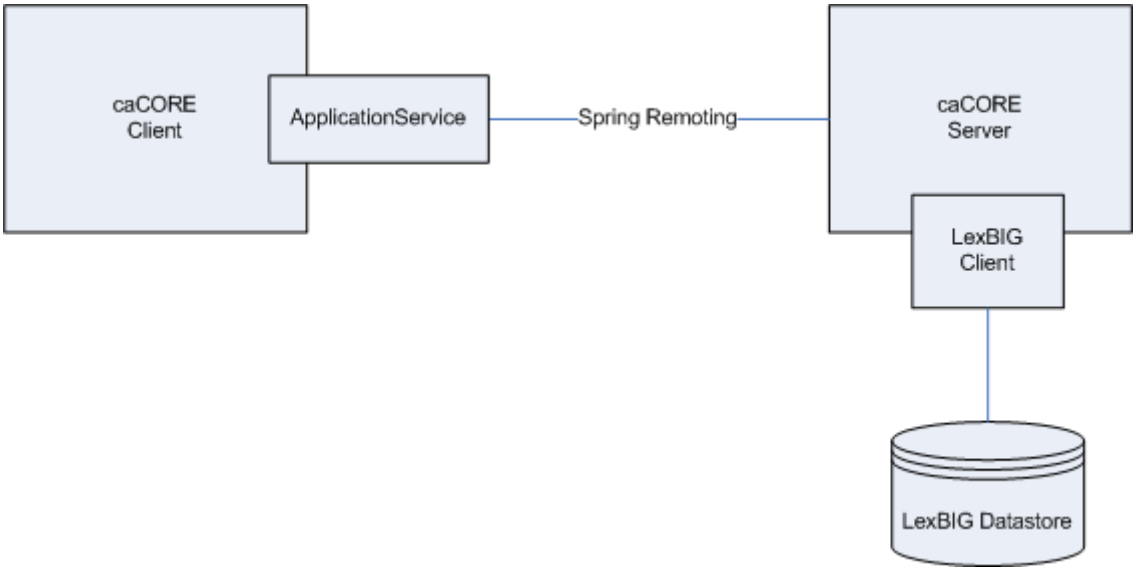

Figure 5-5 DLB Architecture

The DLB environment will be configured on the caCORE EVS Server (http://evsapi.nci.nih.gov/evsapi40). This will give the server access to the LexBIG database and other resources. The client must therefore go through the caCORE EVS server to access any LexBIG data.

LexBIG Annotations

To deal with LexBIG DAOs, integration of the LexBIG API incorporated the addition of Java Annotations marking methods that are safe to execute on the client side and classes that may be passed to the client without being wrapped by a proxy. The annotation is named @lgClientSideSafe. Every method in the LexBIG API that has been made accessible to the caCORE EVS user had to be considered and annotated if necessary.

Aspect Oriented Programming Proxies

LexBIG integration with caCORE EVS was accomplished using Spring Aspect Oriented Programming (AOP) to proxy the LexBIG classes and intercept calls to their methods (Figure 5-6). The caCORE EVS Client wraps every object returned by the LexBIGService inside an AOP Proxy with advice from a LexBIGMethodInterceptor ("the Interceptor").

The "Interceptor" is responsible for intercepting all client calls on the methods in each object. If the method is marked with the @lgClientSideSafe annotation, it proceeds normally. Otherwise, the object, method name and parameters will are sent to the caCORE EVS server for remote execution.

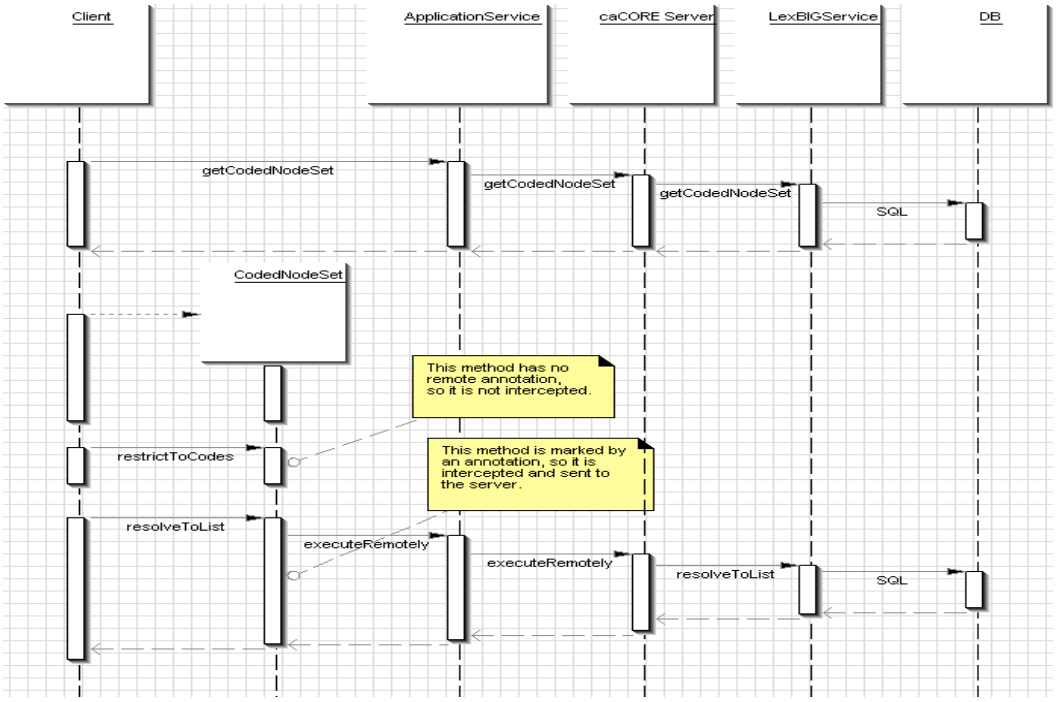

Figure 5-6 Sequence Diagram Showing Method Interception

Once the method is executed remotely, the result is delivered back to the client. If the result is a LexBIG class (i.e. part of the org.LexGrid package) then the Interceptor checks for the @lgClientSideSafe annotation on the resulting class. If the annotation is not found, the class is wrapped in a proxy so that calls to its methods are also remote if necessary.

LexBIG API Documentation

The LexBIG 2.1.1 API was written by the Mayo Clinic. Documentation describing the LexBIG Service Model is available on the LexGRID Vocabulary Services for caBIG GForge site (https://gforge.nci.nih.gov/frs/?group_id=14).

LexBIG Installation and Configuration

The DLB API is strictly a Java interface and requires internet access for remote connectivity to the caCORE EVS server.

In order to successfully access the DLB API, one needs access to the *evsapi-client.jar* available for download on the NCICB website (EVS 3.2 Java API on page 35). The evsapi-client.jar file needs to be available in the system classpath.

Example of Use

The following code sample shows use of the DLB API to retrieve the list of available coding schemes in the LexBIG repository.

```
public class Test {
   /**
    * Intialize program variables
    */
```

```
private String codingScheme = null;
private String version = null;

DLBAdapter adapter = null;
LexBIGService lbSvc;


public Test(String codingScheme, String version)
{
     // Set the EVS URL (for remote access)
     String evsUrl ="http://evsapi.nci.nih.gov/evsapi40/http/
     remoteService";

     boolean isRemote = true;
     this.codingScheme = codingScheme;
     this.version = version;

     // Get the LexBIG service reference from EVS Application Service
     lbSvc = EVSApplicationService.getRemoteInstance(evsUrl);

     // Set the vocabulary to work with
     Boolean retval = adapter.setVocabulary(codingScheme);

     codingSchemeMap = new HashMap();
     try {
        // Get the LexBIG service reference from the RemoteServerUtil
        lbSvc = RemoteServerUtil.createLexBIGService();

        // Using the LexBIG service, get the supported coding schemes
        CodingSchemeRenderingList csrl =
        lbSvc.getSupportedCodingSchemes();

        // Get the coding scheme rendering
        CodingSchemeRendering[] csrs =
        csrl.getCodingSchemeRendering();

        // For each coding scheme rendering....
        for (int i=0; i<csrs.length; i++)  {
           CodingSchemeRendering csr = csrs[i];

           // Determine whether the coding scheme rendering is active
           or not
           Boolean isActive = csr.getRenderingDetail().
           getVersionStatus().
              equals(CodingSchemeVersionStatus.ACTIVE);
           if (isActive != null && isActive.equals(Boolean.TRUE)) {
              // Get the coding scheme summary
              CodingSchemeSummary css = csr.getCodingSchemeSummary();

              // Get the coding scheme formal name
              String formalname = css.getFormalName();

              // Get the coding scheme version
              String representsVersion = css.getRepresentsVersion();
              CodingSchemeVersionOrTag vt = new
              CodingSchemeVersionOrTag();
              vt.setVersion(representsVersion);
```

```
                // Resolve coding scheme based on the formal name
                CodingScheme scheme = null;
                try {
                   scheme = lbSvc.resolveCodingScheme(formalname, vt);
                   if (scheme != null)
                   {
                      codingSchemeMap.put((Object) formalname, (Object)
                      scheme);
                   }
                } catch (Exception e) {
                    // Resolve coding scheme based on the URN
                   String urn = css.getCodingSchemeURN();
                   try {
                      scheme = lbSvc.resolveCodingScheme(urn, vt);
                      if (scheme != null)
                      {
                         codingSchemeMap.put((Object) formalname, (Object)
                         scheme);
                      }
                   } catch (Exception ex) {
                   String localname = css.getLocalName();
                  // Resolve coding scheme based on the local name
                   try {
                      scheme = lbSvc.resolveCodingScheme(localname, vt);
                      if (scheme != null)
                      {
                         codingSchemeMap.put((Object) formalname, (Object)
                         scheme);
                      }
                   } catch (Exception e2) {
                   }
                }
             }
          }
        }
        } catch (Exception e) {
           e.printStackTrace();
        }
     }

     /**
      * Main
      */
      public static void main(String[] args)
      {
        String name = "NCI_Thesaurus";
        String version = "06.12d";

        // Instatiate the Test Class
        Test test = new Test(name, version);
      }
}
```

Distributed LexBIG Adapter

The DLB Adapter is an extension of the DLB API. It is a set of convenience methods to simplify and/or make familiar, use of the DLB API. Access is achieved first through the EVS API and the DLB API. The Javadocs for the DLB Adapter are available at http://gforge.nci.nih.gov/frs/download.php/2704/evsapi4.0_JavaDocs.zip.

Example of Use

The DLB Adapter is designed with dual mode functionality: a direct interface to a local installation of the LexBIG API *('local')* or to be used as an additional layer to the NCI DLB API *('remote')*. It is bundled as part of the evsapi-client*.jar* file and once this *.jar* file is in the application classpath, the programmer can decide how he/she intends to interface with LexBIG (locally or remotely). The code fragment below represents how to use DLB Adapter in both local and remote modes.

```
public class Test {
   /**
    * Intialize program variables
    */
    private String codingScheme = null;
    private String version = null;

    DLBAdapter adapter = null;
    LexBIGService lbSvc;

   /**
    * Test Constructors
    */
    public Test(String codingScheme)
    {
   this.codingScheme = codingScheme;
   this.version = "";
    }

    public Test(String codingScheme, String version)
    {

      /**
       * Establish a reference to the EVS Production URL (remote access).
       */
      String evsUrl =
      "http://evsapi.nci.nih.gov/evsapi40/http/remoteService";

      /**
       * Set the isRemote (LexBIG installation) variable
       */
        boolean isRemote = true;
         this.codingScheme = codingScheme;
         this.version = version;

      /**
      * The DLB Adapter allows you to connect to a co-located/local
      installation of LexBIG or a
       * remote installation of LexBIG.
```

```
         */
          if (isRemote)
          {
         lbSvc = EVSApplicationService.getRemoteInstance(evsUrl);
            adapter = new DLBAdapter((EVSApplicationService)lbSvc);

   System.out.println("\n*************************************************
   *****************");
         System.out.println("\n*****              Instantiate
   EVSApplicationService            *****");

   System.out.println("\n*************************************************
   *****************");
      }
      else
      {

   System.out.println("\n*************************************************
   *****************");
         System.out.println("\n*****              Instantiate
   LexBIGServiceImpl                *****");

   System.out.println("\n*************************************************
   *****************");
         adapter = new DLBAdapter();
      }

         /**
          * Set the vocabulary to the desired coding scheme.
          */
      Boolean retval = adapter.setVocabulary(codingScheme);
      }


     /**
      * testGetTree()
      */
      public void testGetTree()
      {
      String rootname = "C25218";
      boolean direction = true;
      int levels = -1;
      boolean isaflag = true;
      Vector rolenames = new Vector();
      rolenames.add("Technique_Has_Sample_Or_Specimen_Anatomy");


         /**
          * Call the getTree() method passing:
          * 1. The root name
          * 2. The direction
          * 3. The number of levels of depth to return
          * 4. The valid rolenames
          */
      DefaultMutableTreeNode dmtn = adapter.getTree(rootname,
                                          direction,
                                          levels,
```

```
                                        0,
                                    isaflag,
                                    rolenames);
      /**
       * Print the resulting tree.
       */
   adapter.printTree(resultFile, dmtn);
    }


  /**
   * testMetadata()
   */
    public void testMetadata()
    {

            /**
             * Set the name of the desired terminology
             */
       String urn = "NCI_Thesaurus";

            /**
             * Involke the getMetadataProperties() method passing the
terminoloy name (URN).
             * Returns a NameAndValue array.
             */
       NameAndValue[] nv_array = adapter.getMetadataProperties(urn);

            /**
             * Loop through each returned array element and print the name
and content values.
             */
       for (int j=0; j<nv_array.length; j++)
       {
         NameAndValue nv = (NameAndValue) nv_array[j];
         System.out.println("CS: " + urn + "   Metadata Name: " +
nv.getName() + "   Metadata Value: " + nv.getContent());
       }
   }

  /**
   * testMetadata()
   */
    public void testTreeTraversal(String codingSchemeName)
    {

      /**
       * Get ALL root concepts
       */
        CodedEntry[] a = adapter.getRootConcepts();

      /**
       * Loop through each returned value and print.
       */
        for (int i=0; i<a.length; i++)
        {
       CodedEntry ce = (CodedEntry) a[i];
```

```
        printNode(ce, 0);
    }
     }


   /**
    * printNode()
    */
     public void printNode(CodedEntry ce, int level)
     {

       /**
        * Get the concept code for the coded entry
        */
    String code = ce.getConceptCode();

       /**
        * Get the subconcepts for the concept code
        */
    Vector subconcepts = adapter.getSubConcepts(code, true, true);

       /**
        * Loop through each of the subconcepts
        */
    for (int j=0; j<subconcepts.size(); j++)
    {
        String subconceptcode = (String) subconcepts.elementAt(j);

            /**
             * Execute the findConceptByCode() method and print the node.
             */
        CodedEntry sub = adapter.findConceptByCode(subconceptcode, false);
        printNode(sub, level+1);
    }
     }


   /**
    * testMeta()
    */
     public void testMeta()
     {
     adapter.setVocabulary("NCI MetaThesaurus");
     String code = "MTHU000096";
     String source = "LNC";

          /**
           * Get properties by code
           */
     Vector v = adapter.getPropertiesByCode("CL347198");

          /**
           * Find concepts with source code matching...
           */
     Vector v = adapter.findConceptsWithSourceCodeMatching(source, code,
1);
```

```
         /**
          * Find coded entries with source code matching...
          */
    Vector v = adapter.findCodedEntriesWithSourceCodeMatching("NCI
MetaThesaurus", "LNC", "MTHU000096", 1);
    if (v != null)
    {
             /**
              * Loop through each of the coded entries and print the
concept code and name.
              */
        for (int i=0; i<v.size(); i++)
        {
      CodedEntry ce = (CodedEntry) v.elementAt(i);
      System.out.println(ce.getConceptCode());
        }
    }
    }


  /**
   * testSearchConcepts()
   */
    public void testSearchConcepts()
    {
   int limit = 1000;
   String source = "MDR";
   boolean cui = false;
   boolean shortResult = false;
   boolean score = false;
   String scheme = "NCI MetaThesaurus";
   String s = "artery";

        try {

           /**
            * Search all concepts in the MetaThesaurus where the source is
MDR, the search term is 'artery'.
            */
          CodedEntry[] a = adapter.searchConcepts(scheme, s, limit,
source, cui, shortResult, score);


               /**
                * Loop through each of the coded entries and print the
concept code and name.
                */
      for (int i=0; i<a.length; i++)
      {
         CodedEntry ce = (CodedEntry) a[i];
         int j = i+1;
         System.out.println("(" + j +") " + ce.getConceptCode() + "[" +
adapter.getName(ce) + "]");
      }

       } catch (Exception e) {
      e.printStackTrace();
```

```
        }
     }
      }


    /**
     * Main
     */
      public static void main(String[] args)
      {
     String name = "NCI_Thesaurus";
     String version = "06.12d";

     // Instantialte the Test class
     Test test = new Test(name, version);

     // Eexcute the testGetTree() method
          test.testGetTree();

     // Eexcute the testSearchConcepts() method
          test.testSearchConcepts();

     // Eexcute the testMetadata() method
          test.testMetadata();

      }
}
```

Appendix A References

Articles

1. The Description Logic Handbook. Franz Baader, et al. (eds.). Cambridge University Press, 1993.
2. Artificial Intelligence. Patrick Winston. Addison-Wesley, 1984.
3. Artificial intelligence. Minsky M, Hillis D, Rudisch G. New England Journal of Medicine. 1980 Jun 26;302(26):1482.
4. Java Programming: http://java.sun.com/learning/new2java/index.html
5. Extensible Markup Language: http://www.w3.org/TR/REC-xml/
6. XML Metadata Interchange: http://www.omg.org/technology/documents/formal/xmi.htm

caBIG Material

1. caBIG: http://cabig.nci.nih.gov/
2. caBIG Compatibility Guidelines: http://cabig.nci.nih.gov/guidelines_documentation

caCORE Material

1. NCICB: http://ncicb.nci.nih.gov/NCICB/infrastructure
2. caCORE: http://ncicb.nci.nih.gov/NCICB/infrastructure
3. caBIO: http://ncicb.nci.nih.gov/NCICB/infrastructure/cacore_overview/caBIO
4. caDSR: http://ncicb.nci.nih.gov/NCICB/infrastructure/cacore_overview/cadsr
5. CSM: http://ncicb.nci.nih.gov/NCICB/infrastructure/cacore_overview/csm

Software Products

1. Java: http://java.sun.com
2. Ant: http://ant.apache.org/

Appendix B Additional Examples

This appendix provides two additional code examples.

Find Tree For Concept and Association

This example shows how to locate a particular concept node and given association in the identified vocabulary hierarchy.

```
/*
 * Copyright: (c) 2004-2007 Mayo Foundation for Medical Education and
 * Research (MFMER). All rights reserved. MAYO, MAYO CLINIC, and the
 * triple-shield Mayo logo are trademarks and service marks of MFMER.
 *
 * Except as contained in the copyright notice above, or as used to
   identify
 * MFMER as the author of this software, the trade names, trademarks,
   service
 * marks, or product names of the copyright holder shall not be used in
 * advertising, promotion or otherwise in connection with this software
   without
 * prior written authorization of the copyright holder.
 *
 * Licensed under the Eclipse Public License, Version 1.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *       http://www.eclipse.org/legal/epl-v10.html
 *
 * Modified By: NCI CBIIT
 */

package org.LexGrid.LexBIG.example;

import org.LexGrid.LexBIG.DataModel.Collections.AssociationList;
import org.LexGrid.LexBIG.DataModel.Collections.NameAndValueList;
import
org.LexGrid.LexBIG.DataModel.Collections.ResolvedConceptReferenceList;
import org.LexGrid.LexBIG.DataModel.Core.AssociatedConcept;
import org.LexGrid.LexBIG.DataModel.Core.Association;
import org.LexGrid.LexBIG.DataModel.Core.CodingSchemeSummary;
import org.LexGrid.LexBIG.DataModel.Core.CodingSchemeVersionOrTag;
import org.LexGrid.LexBIG.DataModel.Core.NameAndValue;
import org.LexGrid.LexBIG.DataModel.Core.ResolvedConceptReference;
import org.LexGrid.LexBIG.Exceptions.LBException;
import org.LexGrid.LexBIG.Impl.LexBIGServiceImpl;
import org.LexGrid.LexBIG.LexBIGService.LexBIGService;
import org.LexGrid.LexBIG.LexBIGService.CodedNodeSet.PropertyType;
import org.LexGrid.LexBIG.Utility.ConvenienceMethods;
import org.LexGrid.commonTypes.EntityDescription;
```

```
import gov.nih.nci.system.applicationservice.*;


/**
 * Example showing how to determine a branch of associations, starting
 from a
 * specific concept code.
 */
public class FindTreeForCodeAndAssoc {
   final static int MAX_DEPTH = 5;
   public FindTreeForCodeAndAssoc() {
      super();
   }

   /**
    * Entry point for processing.
    * @param args
    */
      public static void main(String[] args) {
      if (args.length < 2) {
         System.out.println(
            "Example: FindTreeForCodeAndAssoc \"C25762\" \"hasSubtype\"");
         return;
      };

      try {
         String code = args[0];
         String relation = args[1];
         new FindTreeForCodeAndAssoc().run(code, relation);
      } catch (Exception e) {
         Util.displayAndLogError("REQUEST FAILED !!!", e);
      }
   }

   public void run(String code, String rel)throws LBException{
      String evsUrl =
      "http://evsapi.nci.nih.gov/evsapi40/http/remoteService";
      CodingSchemeSummary css = Util.promptForCodeSystem();
      if (css != null) {
            LexBIGService lbSvc =
EVSApplicationService.getRemoteInstance(evsUrl);
         String scheme = css.getCodingSchemeURN();
         CodingSchemeVersionOrTag csvt = new CodingSchemeVersionOrTag();
         csvt.setVersion(css.getRepresentsVersion());
         print(code, rel,0, lbSvc, scheme, csvt);
      }
   }

   /**
    * Handle one level of the tree, and recurse to the maximum depth.
    * @param code
    * @param relation
    * @param depth
    * @param lbSvc
    * @param csvt
    * @param scheme
    * @param tagOrVersion
```

```
     * @throws LBException
     */
    protected void print(String code, String relation, int depth,
LexBIGService lbSvc, String scheme, CodingSchemeVersionOrTag csvt) throws
LBException {
       // Perform the query ...
       NameAndValue nv = new NameAndValue();
       NameAndValueList nvList = new NameAndValueList();
       nv.setName(relation);
       nvList.addNameAndValue(nv);

       ResolvedConceptReferenceList matches =
          lbSvc.getNodeGraph(scheme, csvt, null)
             .restrictToAssociations(nvList, null)
             .resolveAsList(
                ConvenienceMethods.createConceptReference(code, scheme),
                true, false, 1, 1,
                null, new PropertyType[] {PropertyType.PRESENTATION},
                null, 1024);

       // Analyze the result ...
       if (matches.getResolvedConceptReferenceCount() > 0) {
          ResolvedConceptReference ref =
             (ResolvedConceptReference)
             matches.enumerateResolvedConceptReference().nextElement();

          // Indent according to level
          StringBuffer sb = new StringBuffer();
          for (int i = 0; i < depth-1; i++)
             sb.append('\t');

          // Print the associations
          AssociationList sourceof = ref.getSourceOf();
          Association[] associations = sourceof.getAssociation();
          for (int i = 0; i < associations.length; i++) {
             Association assoc = associations[i];
             AssociatedConcept[] acl =
             assoc.getAssociatedConcepts().getAssociatedConcept();
             for (int j = 0; j < acl.length; j++) {
                // Print
                AssociatedConcept ac = acl[j];
                EntityDescription ed = ac.getEntityDescription();
                Util.displayMessage(
                   "\t\t" + ac.getConceptCode() + "/"
                      + (ed == null?
                            "**No Description**":ed.getContent()));

                // Recurse
                if (depth < MAX_DEPTH)
                   print(ac.getConceptCode(), relation, depth+1, lbSvc ,
                   scheme, csvt);
             }
          }
       }
    }
```

Search MetaThesaurus for a Particular Concept/Search Term

This example has been commonly used by the caDSR Team. It shows how to search the Metathesaurus for a desired concept.

```
import java.io.File;
import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Enumeration;
import java.util.List;
import java.util.Vector;
import java.util.jar.JarEntry;
import java.util.jar.JarFile;

import gov.nih.nci.system.client.ApplicationServiceProvider;
import gov.nih.nci.system.applicationservice.*;
import gov.nih.nci.evs.query.*;
import gov.nih.nci.evs.domain.*;
import gov.nih.nci.evs.security.*;



public class TestcaDSR
{
   public static void main(String args[])
   {
      TestcaDSR client = new TestcaDSR();
      try
      {
         client.testSearch();
      }
      catch(Exception e)
      {
         e.printStackTrace();
      }
   }

   public void testSearch() throws Exception
   {
      String termStr = "agent";
      int metaLimit=100;
      String metaSource = "*";


      EVSApplicationService evsService =
      (EVSApplicationService)ApplicationServiceProvider.getApplicationServ
      ice();

      try
      {
         EVSQuery evsQuery = new EVSQueryImpl();
         evsQuery.searchMetaThesaurus(termStr, metaLimit, metaSource,
         false, false, false);

         List metaResults = (List)evsService.evsSearch(evsQuery);
```

```
            System.out.println("\n Running updated version....");
            System.out.println("\n Results count = " + metaResults.size());
            for (int i=0; i < metaResults.size(); i++) {
               MetaThesaurusConcept metaConcept =
               (MetaThesaurusConcept)metaResults.get(i);
               if ( metaConcept != null ) {
                  String conceptName = metaConcept.getName();
                  System.out.println("\n\t Concept Name = " + conceptName);
                  String conceptID = metaConcept.getCui();
                  System.out.println("\n\t Concept ID = " + conceptID);
                  ArrayList semantic =
                  metaConcept.getSemanticTypeCollection();
                  System.out.println("\n Semantic size = " +
                  semantic.size());
                  for (int ii=0; ii < semantic.size(); ii++) {
                     System.out.println("\n\t Semantic = " +
                     (SemanticType)semantic.get(ii));
                  }
               }
            }

         } catch(Exception e) {
            System.out.println(">>>"+e.getMessage());
               e.printStackTrace();
         }
      }

}
```

Glossary

The following table contains a list of terms used in this document, with accompanying definitions.

Term	Definition
AJAX	Asynchronous JavaScript and XML
AL	Attributive Language description logic
BioPortal	NCBO's web browser for LexBIG
caBIG	
caBIO	Model and architecture that is the primary programmatic interface to caCORE.
caCORE	Cancer Common Ontologic Representation Environment
caDSR	Metadata registry, based upon the ISO/IEC 11179 standard, used to register the descriptive information needed to render cancer research data reusable and interoperable.
CLM	Common Logging Module. Provides a separate service under caCORE for Audit and Logging Capabilities.
CSM	Common Security Module. Provides a flexible solution for application security and access control for caCORE.
CUI	Concept Unique Identifier
CUI	Concept Unique Identifier
DAML	Agent Markup Language
DARPA	Defense Advanced Research Projects Agency
DL	Description Logic. Family of systems that is especially well suited to the development of ontologies, taxonomies, and controlled vocabularies.
DTS	A proprietary server that does not allow exposure of the API.
EVS	Enterprise Vocabulary Services project. A collaborative effort of the NCI Center for Bioinformatics.
EVS	Enterprise Vocabulary Services project. A collaborative effort of the NCI Center for Bioinformatics.
FL	Frame Language description logic
FOL	First-Order Predicate Logic
GO	Gene Ontology™ Consortium

Term	Definition
ICD-O-3	Additional external vocabularies to NCI Metathesaurus.
IDE	Integrated Development Environment
LexBIG	Open source public domain terminology server LexBIG, developed by the Mayo Clinic as part of the caBIG Program.
MDA	Model Driven Architecture
MedDRA	Additional external vocabularies to NCI Metathesaurus.
MO	Native MGED Ontology is edited in OilEd and distributed in the Defense Advanced Research Projects Agency, Agent Markup Language + Ontology Inference Layer XML format.
NCBO	National Center for Bioontologies
NCI Metathesaurus	Based on NLM's Unified Medical Language System Metathesaurus (UMLS) supplemented with additional cancer-centric vocabulary.
NCI Thesaurus	A biomedical thesaurus developed by EVS in response to a need for consistent shared vocabularies among the various projects and initiatives at the NCI as well as in the entire cancer research community.
NCICB	NCI Center for Bioinformatics
OBO	Open Biomedical Ontologies
OIL	Ontology Inference Layer
OLLT	Obsolete Lower Level Terms
OWL	Web Ontology Language
RDF	Resource Description Framework
RRF	Rich Release Format
SDK	caCORE Software Development Kit or caCORE SDK, a data management framework designed for researchers who need to be able to navigate through a large number of data sources. caCORE SDK is NCICB's platform for data management and semantic integration, built using formal techniques from the software engineering and computer science communities.
SNOMED	Additional external vocabularies to NCI Metathesaurus.
SOAP	Simple Object Access Protocol
SOC	System Organ Class
SSC	Special Search Categories
SUI	Each unique concept name or string in the Metathesaurus has a String Unique Identifier.

Term	Definition
TDE	Terminology Development Environment
UML	Unified Modeling Language
UMLS	NLM's Unified Medical Language System Metathesaurus
UMLS Semantic Network	An independent construct whose purpose is to provide consistent categorization for all concepts contained in the UMLS Metathesaurus, and to define a useful set of relationships among these concepts.
W3C	World Wide Web Consortium
XML	Extensible Markup Language

Index

N

O

Q

R

S

U

V

W

X